DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom – Informatiker

Thema

Portierungsstrategie für ein Hardware-steuerungsprogramm unter Anwendung von Reverse Engineering Techniken

von

René Harder

Alexander Paschold

Betreuer

Prof. Dr. Klaus Bothe

Berlin im August 2003

Institut für Informatik
Math.-Naturwiss. Fakultät II
Lehrstuhl Softwaretechnik
Rudower Chaussee 25
Humboldt-Universität zu Berlin

Erklärungen

Selbständigkeitserklärung

Hiermit erklären wir, dass die vorliegende Diplomarbeit ausschließlich unter Verwendung der angegebenen Quellen selbständig von uns erstellt wurde.

Berlin, den 12.08.2003

René Harder Alexander Paschold

Einverständniserklärung

Hiermit erklären wir unser Einverständnis mit der öffentlichen Ausstellung unserer Diplomarbeit in der Bibliothek des Instituts für Informatik der Humboldt Universität zu Berlin.

Berlin, den 12.08.2003

René Harder Alexander Paschold

Zusammenfassung

Untersuchungsgegenstand der vorliegenden Arbeit ist ein Hardwaresteuerungsprogramm für den Einsatz an Meßplätzen in der Physik der Humboldt Universität Berlin. Für dieses ursprünglich für Windows 3.11 entwickelte Softwaresystem ergab sich im Zuge des technologischen Fortschritts die Forderung auch auf modernen Rechnern mit Windows 2000 eingesetzt zu werden.

Um eine solche Programmtransformation erfolgreich durchzuführen wurde ein detailliertes Vorgehensmodell für die Transformation ähnlich gelagerter Fälle entworfen und am Untersuchungsgegenstand nachvollzogen.

Bei der Transformation des Untersuchungsgegenstandes werden die einzelnen Schritte von der Analyse über die eigentliche Transformation bis zum Test durchlaufen, mit dem Ergebnis einer auf Windows 2000 adaptierten Version. Für die Analysen des Untersuchungsgegenstandes wurde auf Techniken des Reverse Engineerings zurückgegriffen.

Im Rahmen dieser Adaption wurde die Erstellung von Windows 2000-konformen Gerätetreibern erforderlich, so daß eine Treibersuite entstand, die sowohl aktuellen Anforderungen gerecht wird als auch zukünftigen Weiterentwicklungen Raum bietet.

Kapitelübersicht

In der Einleitung wird ein Überblick über den Untersuchungsgegenstand dieser Arbeit gegeben. Dabei werden sowohl der Portierungsgegenstand an sich als auch dessen Anwendungsdomäne und die Motivation für die Portierung vorgestellt.

Im zweiten Kapitel werden die theoretischen Grundlagen für eine Portierung erarbeitet und ein erstes Vorgehensmodell wird konstruiert.

Das dritte Kapitel beschäftigt sich mit der Analyse der im Grundlagenkapitel ermittelten Rahmenbedingungen für den Portierungsgegenstand.

Die Analyse des Portierungsgegenstandes bezüglich der Rahmenbedingungen Entwicklungsumgebung und Softwareumgebung erfolgt in den Kapiteln vier und fünf. Daran anschließend werden die notwendigen Transformationen am Portierungsgegenstand durchgeführt.

Im sechsten Kapitel wird der transformierte Portierungsgegenstand ausführlichen Tests unterzogen. Vorher werden die dafür nötigen theoretischen Grundlagen diskutiert und die Testfälle spezifiziert.

Daran schließt sich ein Kapitel der Fehlerbeschreibungen an. Hier werden Fehler, dokumentiert die während der Portierung und der Tests entdeckt wurden. Sofern möglich werden auch die entsprechenden Fehlerbereinigungen aufgeführt.

Die Zusammenfassung gibt einen komprimierten Überblick über das in dieser Arbeit erreichte und diskutiert dieses. Dabei werden die Aufwendungen der durchgeführten Portierung umrissen.

Das abschließende Kapitel zeigt Ausblicke auf, ausgehend vom erreichten Status des portierten Systems und weist auf mehr und weniger dringliche Entwicklungsschritte hin (korrigieren verbliebener, nicht in den Rahmen der Portierung fallender Mängel).

In den Anhängen finden sich neben Tabellen und Testsequenzen, Dokumentationen die für den Umgang und das Verständnis der Treibersuite elementar sind (Installationsanleitungen etc.) .

Aufgabenteilung

Die Portierung eines umfangreichen Systems wie es XCTL darstellt, bedarf einer strukturierten Vorgehensweise. Das Ausarbeiten einer solchen sowie die Umsetzung dieser in einem Projekt erfordern sowohl koordinierte Arbeitsteilung als auch teamorientierte Zusammenarbeit.

Alexander Paschold erarbeitete die theoretischen Grundlagen die zur Aufstellung der Rahmenbedingungen einer Portierung führten. Des weiteren beschäftigte er sich überwiegend mit der Portierung der Entwicklungsumgebung und der Einführung der Treiberarchitektur in XCTL, sowie mit den übrigen Adaptionen, die für die Portierung nach Windows 2000 nötig waren. Die Überarbeitung der vorhandenen Testsequenzen viel ebenfalls in sein Ressort.

René Harder beschäftigte sich hauptsächlich mit den Grundlagen des Reverse Engineerings, den Grundlagen für die Tests, den Grundlagen der Treiberprogrammierung unter Windows 2000 und der damit einhergehenden Analyse des Zielbetriebssystems. Er erarbeitete darauf aufbauend die gesamte XCTL –Treibersuite (Erstellung der notwendigen Spezifikationen, Implementation von Treibern und Installationsumgebung, Treibertests).

Die erfolgreiche Umsetzung der Portierung und die Erarbeitung eines Vorgehensmodells, erforderte ein hohes Maß an Kommunikation und Teamarbeit, was besonders die Bereiche der Implementation von Treiberarchitektur und -kommunikation sowie die des Tests des Systems und die daraus resultierende Fehlerbereinigung betraf. An dieser Stelle sei auch die äußerst hilfreiche Kommunikation mit den Mitarbeitern der Physik genannt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Problemstellung

Gegenstand des Projektes ist ein Anwendungsprogramm, das im Institut für Physik der Humboldt-Universität Berlin an Labor-Messplätzen zur Untersuchung von kristallinen Halbleiterstrukturen genutzt wird. Aufgabe des Programms ist die Steuerung des Messplatzes und der zugehörigen Hardware. Dies beinhaltet die Vorbereitung und Durchführung der Messung, inklusive der Erfassung und Auswertung der Messdaten. Die Nutzerinteraktion läuft dabei über die grafische Nutzerschnittstelle der Windows-Betriebssystemfamilie.

Das ursprüngliche Programm wurde in Borland C++ implementiert und umfasst mittlerweile ca. 50.000 Programmzeilen (LOC). Zielplattform war Windows 3.11 und damit eine 16-Bit Umgebung.

Aus Instabilitäten während der Ausführung, Programmfehlern und Erweiterungswünschen seitens der Nutzer ergibt sich die Notwendigkeit einer Überarbeitung dieser Software. Im Rahmen der Überarbeitung soll das Programm in zweifacher Hinsicht portiert werden:

(1) Portierung nach Microsoft Visual C++ 6
Überführung des Borland C++ Projektes in ein Microsoft Visual Studio-Projekt und Eliminierung sämtlicher Borland-spezifischer Programmelemente.

(2) Portierung nach Win32 (Windows 2000 $\Rightarrow$ 32-Bit Umgebung)
Dies beinhaltet sowohl eine Portierung der Programmquellen in eine 32-Bit-Umgebung als auch eine Portierung der Programmmodule zur Ansteuerung der Messhardware in die Windows 2000-Systemarchitektur. Letzteres erfordert die Erstellung zusätzlicher Softwaremodule (Gerätetreiber).

Durch geeignete Testverfahren soll sichergestellt werden, dass die Funktionalität des Zielsystems der Funktionalität des Ausgangssystems entspricht.

Als Grundlage der Portierung soll eine allgemeine Portierungsstrategie für ein hardwarenahes Anwendungsprogramm mit vergleichbaren Rahmenbedingungen entwickelt und am Beispiel von *XCTL* dargelegt werden.

Anwendungsdomäne[1]

Am Institut für Physik der Humboldt-Universität Berlin werden in der Arbeitsgruppe "Röntgenbeugung an Schichtsystemen" unter Prof. Köhler kristalline Strukturen von Halbleitern untersucht. Als physikalisches Hilfsmittel dient dabei Röntgenstrahlung (**X**-Ray), die an den Kristallstrukturen von Halbleitern gestreut und zur Steuerung (**C**on**t**ro**l**) der Messung benutzt wird. Daher die Bezeichnung **XCTL-**System.

Das Programm wird an ca. zehn Labor-Messplätzen eingesetzt, die je nach Ausstattung und Messverfahren unterschiedliche Kristalleigenschaften von Halbleiter-Kristallen untersuchen (Topographie, Diffraktometrie, Reflektometrie).

Grundlage dieser Messverfahren ist die Beugung elektromagnetischer Strahlung an den einzelnen Bausteinen des zu untersuchenden Kristalls (Atome, Ionen oder Moleküle). Die gebeugte Strahlung wird anschließend erfasst (z.B. durch einen Detektor oder einen Film) und ausgewertet. Da hierbei die Wellenlänge der Strahlung kleiner als der Abstand der Kristallbausteine sein muss, wird entsprechend kurzwellige Röntgenstrahlung eingesetzt, die durch einen s.g. Kollimator parellelisiert wird. Der Kollimator, die Messprobe und die Detektoren können mit Hilfe von Präzisionsmotoren frei im Raum bewegt werden.

Jeder Messplatz ist mit einem PC verbunden. Aufgabe der darauf laufenden XCTL-Software ist:

(1) die Ansteuerung und Koordinierung der Motoren, Detektoren und Kollimatoren zwecks Justage der Messanlage und die Kontrolle und Steuerung des Messvorganges.

(2) die Erfassung, Transformation und Darstellung der Messdaten.

[1] Quelle: [Projekt98]

Motivation

Von der Migration der Entwicklungsumgebung wird im Wesentlichen eine deutliche Aufwandsreduzierung und Erleichterung der Vorgänge der Programmerweiterung und –analyse erwartet. Diese Erwartung stützt sich vor allem auf eine breitere Werkzeugunterstützung für Microsoft Visual Studio und die scheinbar größere Konformität des zugehörigen Compilers bezüglich des ANSI-C++ Standards.
Bedingt durch die Migration der Entwicklungsumgebung nach Microsoft Visual Studio ergibt sich zwangsläufig die Notwendigkeit der Portierung des XCTL-Systems nach Win32, da Visual Studio in der aktuellen Version lediglich Win32 als Zielplattform unterstützt. Diese auf den ersten Blick nachteilig anmutende Tatsache bringt allerdings auch eine Anzahl von Vorteilen mit sich. So kann XCTL nach abgeschlossener Portierung nach Win32 die Vorteile und Errungenschaften einer modernen 32-Bit-Betriebssystemarchitektur nutzen. Als Beispiele seien hier nur der Speicherschutz, der zu einer signifikant höheren Stabilität des Gesamtsystems führt, und die Verbesserung der grafischen Nutzerschnittstelle genannt.
Nicht unerwähnt bleiben sollte, dass Microsoft seit einiger Zeit den Support für sämtliche Betriebssysteme unterhalb von Windows 2000 eingestellt hat. Das heißt, dass zukünftige Entwicklungen, sowohl technische, als auch sicherheitsrelevante, in diesen Systemen nicht mehr nachvollzogen werden und somit den darauf laufenden Anwendungen auch nicht zur Verfügung stehen.
Mit der Portierung von XCTL nach Windows 2000 als einen typischen Vertreter der Win32-Betriebssystemfamilie eröffnet sich für o.g. Software die Möglichkeit der Nutzung, vor allem der zukünftigen Nutzung, des technologischen Fortschritts.
Zusammenfassend führt die Portierung des XCTL-Systems zu einer zeitgemäßen und verbesserten Erweiter-, Wart- und Nutzbarkeit der Labormessplätze, wovon sowohl die Entwickler als auch die Nutzer von XCTL profitieren.

2. Theoretische Grundlagen der Portierung

Portierung - Begriffsklärung

Unter dem Begriff *Portierung* versteht man allgemein die Übertragung eines Softwaresystems in eine andere Systemumgebung durch Überarbeitung und Änderung der Quellen. Während dieses Vorgangs darf die Korrektheit nicht im geringsten Maße und die Effizienz der zu portierenden Software nur unwesentlich beeinträchtigt werden, so dass dem Anspruch der Spezifikationserhaltung entsprochen wird.[1] Letztere Bedingung kann natürlich nur erfüllt werden, wenn schon das Ausgangssystem spezifikationskonform war.
Ein Softwaresystem ist um so portabler, je geringer der notwendige Adaptierungssaufwand ist bzw. je niedriger die durch den Portierungsvorgang verursachten Kosten ausfallen [Müller02].

Diese Begriffserläuterung kann auch in eine formale Definition umgesetzt werden [Kaindl 88]:

➢[1] Umgebung *E* ist ein Tripel aus einer Menge von Sprachprozessoren *l* (*language processors*), dem Betriebssystem *o* (*operating system*) und der Hardware *m* (*machine*)

$$E = (\{l_1,...,l_n\}, o, m) \text{ mit } \{l_1,...,l_n\}, n \in \mathrm{N}$$

Portierung ist dann die Abbildung

$$(P, E) \rightarrow (P', E')$$

mit dem zu portierenden Programm *P*, der Ausgangsumgebung $E = (\{l_1,...,l_n\}, o, m)$, der Zielumgebung $E' = (\{l'_1,...,l'_n\}, o', m')$ und dem Portierungsergebnis P', unter der Bedingung, dass $(\{l_1,...,l_n\} \neq \{l'_1,...,l'_n\}) \vee (o \neq o') \vee (m \neq m')$. Eine Übertragung ohne Änderung eines der Elemente der Ausgangsumgebung kann somit nicht als Portierung bezeichnet werden.

[1] Die einzig ggf. akzeptable Änderung der Funktionalität wäre eine Funktionserweiterung des Portierungsgegenstandes.

Entsprechend der formalen Begriffsdefinition ergibt sich der Portierungsaufwand aus dem Unterschied von Ausgangsumgebung E und Zielumgebung E'. Als Maß des Portierungsaufwandes kann die Anzahl der zu ändernden Quelltextzeilen angenommen werden.

Im Folgenden soll der Begriff Portierung verstanden werden, als die Erstellung eines Softwaresystems in einer neuen Systemumgebung unter Verwendung und Adaptierung eines bereits bestehenden Systems [Mooney97] mit der Maßgabe, dass die wesentlichen Funktionseigenschaften dieses Softwaresystems nicht verändert werden (Spezifikationserhaltung).

Rahmenbedingungen der Portierung und Portierungsmodell

Als Rahmenbedingungen der Portierung werden im Folgenden Faktoren bezeichnet, die den mit der Portierung verbundenen Aufwand maßgeblich bestimmen.

Einige der Faktoren können direkt aus der formalen Begriffsdefinition der Portierung abgeleitet werden:

(1) Hardware (m)

Die Hardware bildet die Ausführungsbasis des Softwaresystems.
Definiert wird dieser Einflussfaktor zum einen durch die grundlegenden Eigenschaften der ausführenden Hardware, die hauptsächlich durch die spezifische Rechnerarchitektur bestimmt werden. Zum anderen fließen auch die Eigenschaften und Anforderungen der vom Portierungsgegenstand ggf. genutzten speziellen Hardware (Hardwareerweiterungen, Peripheriegeräte, ...) ein.

(2) Systemsoftware/Betriebssystem (o)

Die Systemsoftware realisiert eine Abstraktionsebene der Hardware. Sie stellt durch benannte Schnittstellen grundlegende Funktionen zum Betrieb der Hardware zur Verfügung und ermöglicht hierdurch Anwendungsprogrammen einen bestimmten Grad der Hardwareunabhängigkeit (top-down-Sicht). Systemschnittstellen sind

üblicherweise in Form von Funktionsbibliotheken realisiert, die von Applikationen genutzt werden können.
Aufgabe der Systemsoftware ist die Verwaltung aller Elemente eines Hardwaresystems, wie Prozess- und Speicherverwaltung, Ein- und Ausgabe-Verwaltung, Verwaltung der allgemeinen Systemressourcen und Verwaltung der Daten (bottom-up-Sicht).
Definiert wird dieser Einflussfaktor durch Eigenschaften und Umfang der bereitgestellten Funktionsbibliotheken und die spezifische Umsetzung der o.g. Aufgaben.

(3) Entwicklungssystem/Programmiersprache (l)

Aufgabe des Entwicklungssystems ist die Transformation von Programmen von einer menschenlesbaren in eine maschinenlesbare Repräsentation. Zusätzlich werden ggf. über Funktionsbibliotheken abstrakte Lösungsansätze für allgemeine Programmfunktionen bereitgestellt.
Definiert wird dieser Einflussfaktor über die Eigenschaften der zur Verfügung gestellten Programmiersprache und die Eigenschaften und den Umfang der enthaltenen Funktionsbibliotheken.

Da diese drei Faktoren die Umgebung des Softwaresystems definieren und der Aufwand der Adaption hauptsächlich durch den Unterschied zwischen Host- und Zielsystem bestimmt wird, haben sie direkten Einfluss auf den Portierungsaufwand. D.h., je weniger sich Host- und Zielsystem in diesen Faktoren unterscheiden, um so niedriger ist der für die Portierung anzusetzende Aufwand.

Nach [Buschhorn 88] ergibt sich eine weitere Gruppe von Rahmenbedingungen direkt aus dem Portierungsgegenstand. Diese Faktoren werden entsprechend als interne Faktoren bezeichnet und haben einen indirekten Einfluss auf den Portierungsaufwand.

(4) Komplexität des Portierungsgegenstandes

Im Rahmen der Portierung vorgenommene Anpassungen ziehen unter Umständen weitere Quelltextänderungen nach sich. Je höher die Komplexität des Portierungsgegenstandes, um so aufwändiger ist die für

eine korrekte Durchführung der Anpassung notwendige Analyse von Seiteneffekten und die Durchführung entsprechender Verifikationstests.
Der mit der Adaption verbundene Aufwand steht also im direkten Zusammenhang mit der Komplexität des zu portierenden Softwaresystems.

(5) Funktionalität des Portierungsgegenstandes

Entsprechend der Definition des Portierungsbegriffs darf die Funktionalität des Portierungsgegenstandes durch die vorgenommenen Portierungstransformationen nicht im geringsten Maße eingeschränkt werden. Der mit der Einhaltung und Verifikation dieser Bedingung verbundene Aufwand (z.B. Verifikationstests) ist offensichtlich direkt abhängig von der Funktionalität des zu portierenden Softwaresystems.

(6) Quellcode des Portierungsgegenstandes

Da die mit dem Portierungsprozess verbundenen Analyse- und Adaptionsvorgänge direkt mit dem Quelltext des Portierungsgegenstandes verknüpft sind, wirken sich entsprechend auch die spezifischen Quelltexteigenschaften wie Umfang, Gliederung, Stil oder Grad der Kommentierung auf den Aufwand des Portierungsprozesses aus.

Eine letzte Gruppe von Rahmenbedingungen ist gänzlich außerhalb der Systemumgebung angesiedelt. Diese externen Faktoren haben ebenfalls einen indirekten Einfluss auf den mit der Portierung verbundenen Aufwand.

(7) Dokumentation

Idealerweise können aus der Dokumentation der Hostumgebung, der Zielumgebung und schließlich des Portierungsgegenstandes alle für den Prozess der Portierung relevanten Informationen auf einfache Art und Weise ermittelt und somit der Aufwand für die notwendige Analyse relativ klein gehalten werden. Unzureichende Dokumentation wirkt sich entsprechend in einer Erhöhung des Portierungsaufwandes aus, da

notwendige Informationen auf anderem Wege (z.B. Reverse Engineering) beschafft werden müssen.

(8) Werkzeuge

Durch den Einsatz von Portierungswerkzeugen kann ein Teil des Portierungsprozesses automatisiert werden. Eine Erhöhung des Anteils der automatisierten Portierungsschritte führt zu einer Verringerung des Portierungsaufwandes.

(9) Erfahrung des Portierenden

Unabhängig von allen anderen Faktoren wird der Aufwand des Portierungsprozesses maßgeblich durch die Erfahrung des durchführenden Menschen bestimmt. Je besser die Kenntnisse des Portierenden bzgl. der Host- und Zielumgebungen sind, desto geringer ist der für die Portierung notwendige Einarbeitungsaufwand. In früheren Portierungsprojekten gesammelte Erfahrungen führen zur Vermeidung bekannter Fehler und tragen zu einer Steigerung der Effektivität des Portierungsprozesses bei. Die Fähigkeiten des Portierenden sind von um so größerer Bedeutung, je ungeeigneter die zur Verfügung stehende Dokumentation ist.

Die bisherigen Ausführungen werden in dem Portierungsmodell nach [Buschhorn 88] zusammengefasst (Abb. [2.1]).

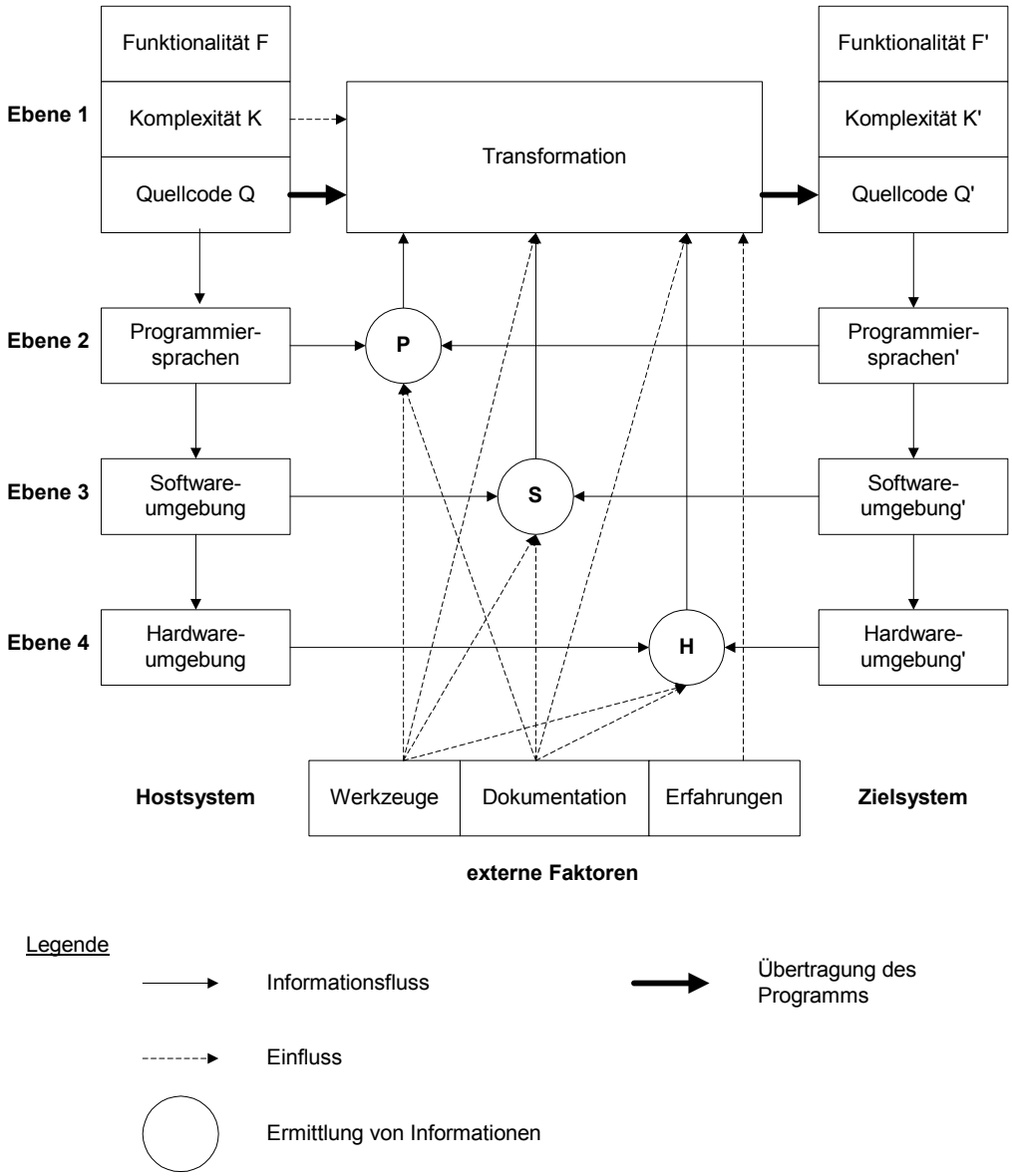


Abbildung [2.1]

Host- und Zielsystem besitzen jeweils spezifische Ausprägungen der auf den Ebenen 1 bis 4 angeordneten Rahmenbedingungen. Bei der Übertragung eines Softwaresystems von der Host- in die Zielumgebung ist nun ein Ausgleich zwischen diesen Zuständen durch Transformation der Programmquellen notwendig (fett gezeichnete Pfeile). Der Aufwand der Transformation wird durch das Maß der Unterschiedlichkeit der jeweiligen Merkmalsausprägungen bestimmt. Entsprechend ist für die Aufwandsbestimmung und Definition der notwendigen Transformationsschritte eine Analyse der spezifischen Ausprägungen der Rahmenbedingung notwendig (dünn gezeichnete Pfeile und Kreise). Die

externen Faktoren wirken indirekt auf die Analyse- und Transformationsprozesse (gestrichelte Pfeile).

Dieses Portierungsmodell kann genutzt werden, um obige formale Definition des Portierungsbegriffs, um die Definition des Begriffs *Programm* und die Definition der Eigenschaften des Transformationsprozesses zu erweitern:

➢[2] Programm *P* ist ein Tripel aus Funktionalität *F*, Komplexität *K* und dem Quellcode *Q*

$$P = (F, K, Q)$$

Transformation ist dann die Abbildung

$$(P) \rightarrow (P')$$

mit dem zu portierenden Programm $P = (F, K, Q)$ und dem Portierungsergebnis $P' = (F', K', Q')$.

Von Portierung wird dann gesprochen, wenn zusätzlich zu den Bedingungen von ➢[1] gilt:

$$F \leq F' \quad und \quad Q \neq Q'$$

Portierungsarten

Im weiteren Verlauf wird der Begriff *Rahmenbedingung* auf die direkten Einflussfaktoren *Entwicklungsumgebung*, *Softwareumgebung* und *Hardwareumgebung* reduziert.

Abhängig von der Art der sich ändernden Umgebungselemente vom Host- zum Zielsystem ist der Portierungsprozess durch unterschiedliche Strategien, Aufwände und relevante Rahmenbedingungen gekennzeichnet.

Im Wesentlichen werden drei Portierungsarten unterschieden:

(1) Änderung der Entwicklungsumgebung (Programmiersprache)

Diese Portierungsart erfordert eine Transformation der Sprachkonstrukte von Host- in Zielsprache. Zusätzlich notwendig ist eine Anpassung der Aufrufe extern bereitgestellter Funktionen (Funktionsbibliotheken).

Grundsätzlich kann zwischen einem vollständigen Wechsel der Implementationssprache und dem Wechsel der Entwicklungsumgebung unter Beibehaltung der ursprünglichen Sprache unterschieden werden. Letzteres verursacht im günstigsten Fall kaum Aufwand, da das Softwaresystem lediglich mit der neuen Entwicklungsumgebung übersetzt werden muss. Maßgeblich hierfür sind der genutzte Sprachumfang der Implementationssprache, die Konformität bezüglich des Sprachstandards und die Verfügbarkeit äquivalenter Funktionsbibliotheken in Host- und Zielsystem.
Der vollständige Wechsel ist mit einem sehr viel größeren Aufwand verbunden, da hier zusätzlich eine Transformation des gesamten Quelltextes vorgenommen werden muss.

(2) Änderung der Systemssoftware/Betriebssystem (Softwareumgebung)

Diese Portierungsart erfordert eine Transformation der Nutzung von Systemschnittstellen durch den Portierungsgegenstand einschließlich der Nutzerschnittstelle.

Der Aufwand der Adaption wird durch das Maß der Unterschiedlichkeit der jeweiligen Systemschnittstellen bezüglich Funktionsumfang und Syntax bestimmt. Wesentlich ist ebenfalls der Grad der Nutzung dieser Schnittstellen durch den Portierungsgegenstand. Ein Umgehen der Systemschnittstellen führt im Allgemeinen zu einer Erhöhung des Portierungsaufwandes.
Zu beachten sind hierbei auch aus internen Funktionsmechanismen ggf. resultierende Einschränkungen des Zielsystems, die aufgrund evtl. zusätzlich notwendiger Modifikationen zur Kompensation dieser Einschränkungen einen negativen Einfluss auf den sich ergebenden Portierungsaufwand ausüben können.

(3) Änderung der Hardwareumgebung

Hier besteht der Portierungsvorgang aus der Adaption der Kommunikationsschnittstelle zwischen Portierungsgegenstand und genutzter Hardware.

Bei der genutzten Hardware sind zwei Klassen zu unterscheiden. Zum einen die Hardware der Ausführungsumgebung und zum anderen die spezielle Hardware des Portierungsgegenstandes. Unter der Hardware der Ausführungsumgebung sind die Basiselemente eines Computers, wie CPU, Hauptspeicher, Massenspeicher, Tastatur usw. zu verstehen, also all jene Dinge, die einen Computer im wesentlichen definieren und grundlegend für die Ausführung von Programmen sind. Diese Hardwarekomponenten werden im Allgemeinen von sämtlicher Software genutzt. Anwendungsspezifische Hardwarekomponenten hingegen besitzen durch ihre spezialisierte Funktionalität einen weit engeren Anwendungsbereich und werden in einem System nur von einer sehr geringen Anzahl von Programmen genutzt. Diese Hardwarekomponenten ermöglichen meist erst die sinnvolle und korrekte Ausführung der entsprechenden Anwendung und stellen logisch betrachtet einen Teil dieser Software dar.

In welchem Maße sich die Änderung der Hardware der Ausführungsumgebung auf den Portierungsaufwand auswirkt, ist nicht nur von der Rahmenbedingung *Hardware* abhängig, sondern wird zu einem großen Teil auch von der Ausprägung der Rahmenbedingung *Entwicklungssystem* bestimmt. Wurde das zu portierende Softwaresystem in einer Hochsprache implementiert, die in gleicher Art auch in der Zielumgebung zur Verfügung steht, sollte der gesamte Portierungsaufwand lediglich in einem neuerlichen Übersetzen des Portierungsgegenstandes in der Zielumgebung bestehen. Liegt das zu portierende Softwaresystem allerdings in Maschinensprache vor, so muss je nach Unterschiedlichkeit der Hardware des Host- und Zielsystems im ungünstigsten Fall der Portierungsgegenstand in der Maschinensprache des Zielsystems komplett neu implementiert werden, was einem sehr hohen Portierungsaufwand entspricht. Der Portierungsaufwand ist also nicht nur direkt abhängig vom Grad der Unterschiedlichkeit der Hardware des Host- und Zielsystems, sondern auch vom Hardwareabstraktionsgrad der Implementationssprache und deren Verfügbarkeit auf dem Zielsystem.

Der durch eine Änderung der speziellen Hardware des Portierungsgegenstandes verursachte Portierungsaufwand wird zum einen durch das Maß der Änderung der entsprechenden Kommunikationsschnittstelle bestimmt und zum anderen durch den Abstraktionsgrad dieser Schnittstelle. Je niedriger das Abstraktionsniveau liegt, also je spezifischer die Ansteuerung der Hardware implementiert wurde, um so höher ist der notwendige Adaptionsaufwand. Der eigentliche Portierungsvorgang besteht dann im Wesentlichen in der Anpassung der Kommunikationsschnittstelle an die Anforderungen der neuen Hardwarekomponenten.

(4) Kombinationen

Ändern sich die Umgebungselemente in mehr als einer Ebene (siehe Portierungsmodell), so führt dies zu einer Kombination der zuvor beschriebenen Portierungsarten sowohl in der Definition der relevanten Rahmenbedingungen als auch der Aufwände und Portierungsstrategien.

Portierungsrelevante Informationen

Neben der genauen Kenntnis des internen Aufbaus des zu portierenden Softwaresystems ist die Analyse der Rahmenbedingungen des Portierungsprozesses eine notwendige Voraussetzung für eine erfolgreiche Adaption des Portierungsgegenstandes. Nur so können notwendige Transformationen definiert und die zu modifizierenden Quelltextfragmente effizient identifiziert und lokalisiert werden. Entsprechend der Art der Portierung sind für den Portierungsprozess verschiedene Rahmenbedingungen bestimmend. Portierungsrelevant sind dann alle Bedingungen/Zustände, welche die entsprechenden Rahmenbedingungen definieren. Für ein konkretes Portierungsprojekt sind nur jene Ausprägungen eines Einflussfaktors von Bedeutung, die tatsächlich Einfluss auf den Portierungsgegenstand ausüben, also z.B. nur die in der Portierungsbasis genutzten Bibliotheksfunktionen und nicht die Gesamtheit der auf Host- bzw. Zielsystem zur Verfügung stehenden Funktionen.

(1) Entwicklungsumgebung

Bestimmt wird diese Rahmenbedingung hauptsächlich durch die spezifischen Eigenschaften der genutzten Implementationssprache und das Maß der Unterstützung des Entwicklungsprozesses durch das Entwicklungssystem. Relevante Faktoren sind also Umfang und Standardkonformität der Implementationssprache sowie Funktionsumfang und Schnittstellen der zur Verfügung gestellten Laufzeit- und Funktionsbibliotheken.

- Syntax der Implementationssprache (Umfang, Besonderheiten, Abweichungen vom Sprachstandard)
- Verfügbarkeit spezieller Spracherweiterungen (Makros, Ausnahmebehandlung, etc.)
- Umfang der bereitgestellten Bibliotheksfunktionen
- Syntax der Funktionsschnittstellen

(2) Systemsoftware/Betriebssystem (Softwareumgebung)

Portierungsrelevante Informationen bezüglich dieser Rahmenbedingung resultieren wesentlich aus den Eigenschaften entsprechender Systemschnittstellen und Funktionsmechanismen und hierin bedingter Einschränkungen und Fähigkeiten.

- Funktion und Syntax der Systemsschnittstellen
- interne Mechanismen (Prozessverwaltung, Speicherverwaltung, etc.)
- Nutzerschnittstelle
- Sicherheitsmechanismen

(3) Hardwareumgebung

Dieser Einflussfaktor wird durch die grundlegenden Eigenschaften der ausführenden (und ggf. speziellen) Hardware und daraus resultierenden Einschränkungen bzw. Fähigkeiten bestimmt.

- Datenbreite
- interne Darstellung von Daten (Byteorder)

- Speicherorganisation
- Anzahl und Verwendung bereitgestellter Register
- Syntax und Umfang der Maschinenbefehle
- Hardwareschnittstellen (z.B. BIOS, Firmware, Kommunikations-register)

Unter Umständen kann dieser Einflussfaktor von den Rahmenbedingungen *Systemumgebung* und *Entwicklungsumgebung* teilweise oder völlig überdeckt werden. Dies ist insbesondere dann der Fall, wenn der Portierungsgegenstand ausschließlich über die von diesen Ebenen bereitgestellte Schnittstellen auf die Hardwareumgebung zugreift.

Allgemeines und spezielles Vorgehensmodell

Auf den bisherigen Erläuterungen aufbauend kann ein allgemeines Modell des Portierungsvorgangs entwickelt werden.

(1) Bestimmen der Ausprägung der Rahmenbedingungen des Host- und Zielsystems.
Hierbei werden Host- und Zielsystem bezüglich der spezifischen Eigenschaften von Entwicklungsumgebung, Softwareumgebung und Hardwareumgebung untersucht und die gewonnenen Informationen dokumentiert.

(2) Bestimmen der Art der Portierung anhand der ermittelten relevanten Rahmenbedingungen.
Als relevante Rahmenbedingungen werden diejenigen angesehen, die sich in ihrer Ausprägung vom Host- zum Zielsystem signifikant ändern. Daraus können dann die Art der Portierung ermittelt (Siehe Portierungsarten) und ...

(3) Ableiten der formalen Transformationsschritte.
... in einem nächsten Schritt die notwendigen formalen Transformationsschritte abgeleitet werden.

(4) Analyse des Portierungsgegenstandes bezüglich der formalen Transformationsschritte.
Um die notwendigen Adaptionen vornehmen zu können, müssen die entsprechenden Quelltextfragmente identifiziert und lokalisiert

werden. Hierfür ist eine Analyse des Portierungsgegenstandes notwendig.

(5) Ableiten der speziellen Transformationsschritte und Durchführen der Transformation.
Auf Basis der gewonnen Informationen werden notwendige Quelltextmodifikationen entworfen und durchgeführt.

(6) Test des Transformationsergebnisses.
Der abschließende Test soll die Erhaltung der Funktionalität feststellen und somit den Erfolg der Portierung dokumentieren.

Von diesem Modell kann unter Beachtung der in Kapitel 1 erläuterten Zielstellung dieser Arbeit ein spezialisiertes Vorgehensmodell abgeleitet werden, das in den folgenden Kapiteln umgesetzt und präzisiert wird (Abbildung [2.2]).

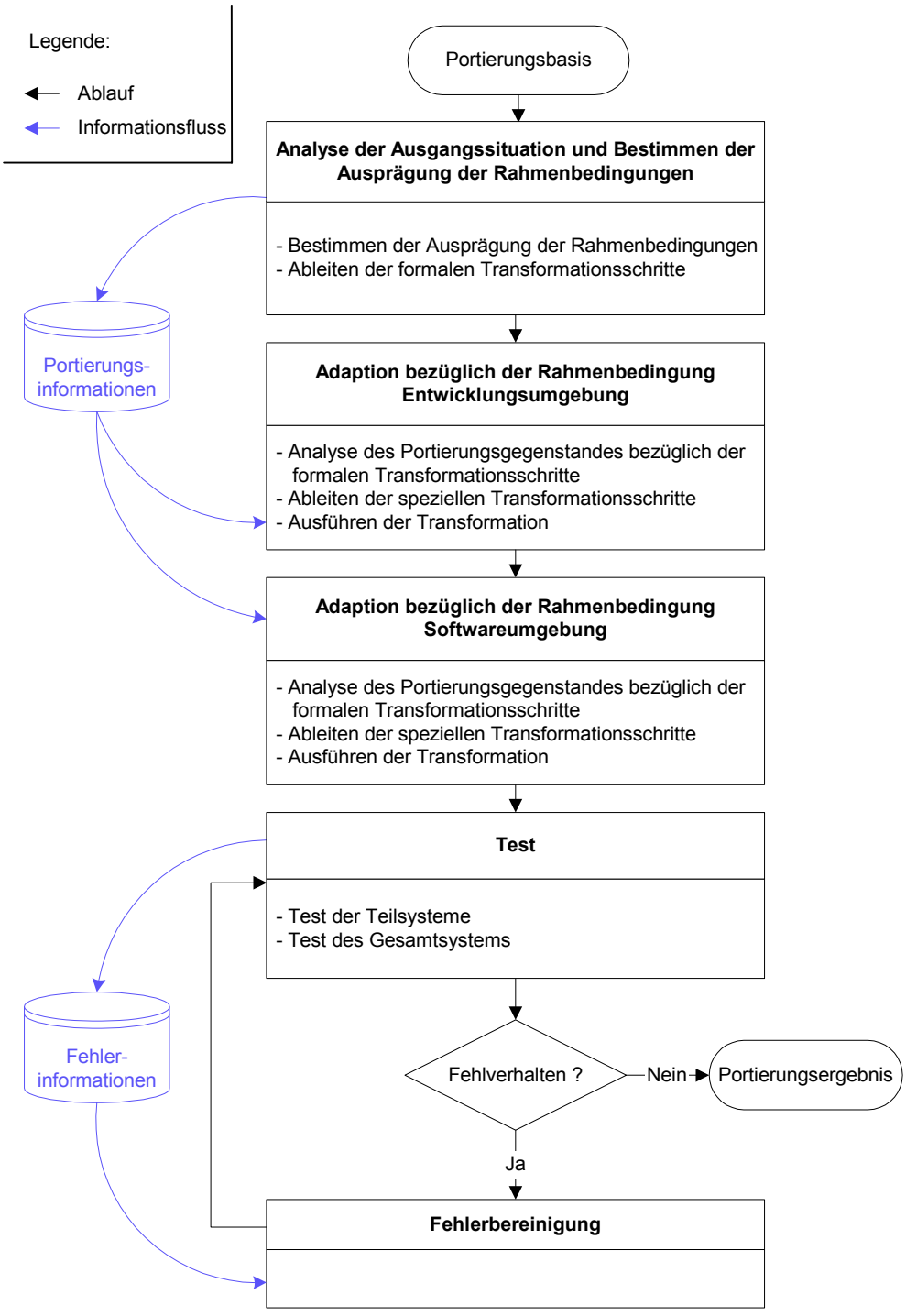


Abbildung [2.2] Vorgehensmodell

Gewinnung portierungsrelevanter Informationen durch Reverse Engineering

Notwendige Voraussetzung einer erfolgreichen Portierung ist die genaue Kenntnis von Architektur und Implementation des Portierungsgegenstandes. Der Portierende muss sich mit den internen Abläufen des Softwaresystems auseinander setzen und diese verstehen, um einerseits portierungsrelevante Quelltextteile zu identifizieren und andererseits notwendige Adaptionen korrekt zu entwerfen und umzusetzen.

Grundlegend für den Portierungsprozess ist somit die Gewinnung von Informationen über die Interna des zu portierenden Softwaresystems, woraus sich zwangsläufig die Frage nach adäquaten Informationsquellen ergibt. Programmdokumentation und Quelltextkommentare sind entsprechende Informationsquellen, jedoch allzu oft nur in unzureichendem Maße vorhanden. Liegt keine geeignete Dokumentation vor, so muss ein anderer Weg der Informationsgewinnung gefunden werden. Eine Möglichkeit ist das s.g. *Reverse Engineering*.

Reverse Engineering - Begriffsklärung

Der Begriff des Reverse Engineering hat seinen Ursprung im Bereich der Analyse von Hardwaresystemen [Balzert98] und bezeichnet im Softwarebereich die Umkehrung des normalen Softwareentwicklungsprozesses, des so genannten *Forward Engineerings*. Während beim Forward Engineering von den spezifizierten Produktanforderungen ausgehend eine Architektur des Softwaresystems entworfen und abschließend implementiert wird, geht man beim Reverse Engineering den umgekehrten Weg von der Implementation zur Spezifikation. Abbildung [2.3] veranschaulicht die Beziehungen zwischen Forward und Reverse Engineering [Klösch95].

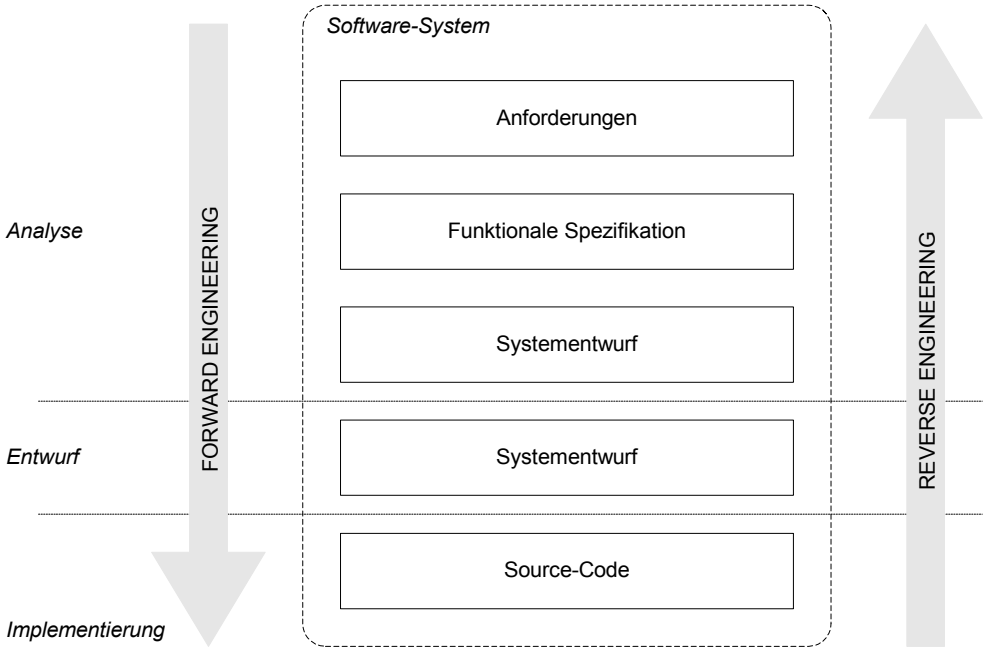


Abbildung [2.3] Forward und Reverse Engineering

Mit Hilfe von Werkzeugen oder durch manuelle Analysen wird versucht, aus den Quellen die Systembeschreibung auf höheren Abstraktionsebenen abzuleiten. Wenn Quelltexte nicht verfügbar sind, können diese durch Dekompilierung bzw. Disassemblierung des zu untersuchenden Softwaresystems ggf. gewonnen werden. Nach [Balzert98] liegt die Hauptaufgabe des Reverse Engineering in der Erstellung der Dokumente, die heute standardmäßig bei einer Softwareneuentwicklung entstehen, aber bei einem Altsystem eben nicht vorhanden sind. Es soll somit das Verständnis des Altsystems verbessert werden, was wiederum hilft, Wartung und Weiterentwicklung zu vereinfachen, bzw. die Grundlage für Änderung und Reimplementation (Reengineering) oder Portierung des Altsystems darstellt.[2]

[2] Da das Reverse Engineering in der vorliegenden Arbeit der Informationsgewinnung für den Zweck der Portierung dient und keine Änderungen der Altsoftware in Funktionalität und Architektur vorgenommen werden soll, wird hier nicht näher auf das Reengineering eingegangen. Dazu sei auf die Literatur [Klösch95], [Balzert98] verwiesen.

Methoden des Reverse Engineering

Wie bereits erwähnt, handelt es sich beim Reverse Engineering um die Analyse eines Softwaresystems bzw. dessen Quelltexte. Dabei unterscheidet man zwischen statischer und dynamischer Analyse sowie der Komplexitätsanalyse (vergl. [Balzert98]). Letztere sei nur der Vollständigkeit halber erwähnt, sie dient mit ihren Maßzahlen der Bewertung der Sanierungsbedürftigkeit einer Software und findet hier entsprechend keine Anwendung.

(1) Statische Analyse

Unter dem Begriff statische Analyse werden sämtliche Aktivitäten zusammengefasst, die ein Softwaresystem auf Grundlage seiner Quelltexte untersuchen.

Methoden der statischen Analyse mit besonderer Bedeutung für die Gewinnung portierungsrelevanter Informationen:

(1.1) lexikalische Analyse
- Identifizierung und Lokalisierung der im Quelltext enthaltenen Sprachelemente

(1.2) semantische Analyse
- liefert Informationen über die Bedeutung bestimmter Quelltextfragmente und -elemente für das Gesamtsystem wie z.B. Schleifen, Funktionen, Variablen

(1.3) syntaktische Analyse
- Untersuchung der grammatikalischen Struktur des Quelltextes

(1.4) Querverweisanalyse
- Erstellung von Verwendungsnachweisen für einzelne Quelltextelemente

Typisches Anwendungsszenario, welches die Verwendung und die Kombination der Analysemethoden an einem konkreten (XCTL-basierten) Einsatzzweck veranschaulicht:

Mit Hilfe der lexikalischen Analyse werden beispielsweise Schleifen im Quelltext lokalisiert, die in einer anschließenden semantischen Analyse dahingehend untersucht werden, ob z.B. systemabhängige Warteschleifen implementiert wurden.
Auch stellt die Suche nach Bezeichnern von Bibliotheksfunktionen oder Strukturen, die bekanntermaßen Portierungsprobleme verursachen (siehe Tabellen [D.2] [D.3]), eine Form der lexikalischen Analyse dar.
Die mit der Querverweisanalyse erstellten Verwendungsnachweise der einzelnen Programmelemente sind besonders hilfreich, um die Aufrufbeziehungen innerhalb des Programms zu erfassen. Über diese kann dann im Rahmen der Portierung z.B. festgestellt werden, auf welche Bereiche im Quelltext eine Änderung der Parameterliste einer Funktion Auswirkungen hat.
Ein willkommener Nebeneffekt dieser Analyse ist das Aufdecken nicht verwendeter Funktionen, da diese in den Verwendungslisten nicht aufgeführt werden[3].

(2) Dynamische Analyse

Die dynamische Analyse umfasst alle Aktivitäten, die sich auf die Untersuchung des Programmverhaltens zur Laufzeit beziehen. Hierzu wird die Reaktion des zu untersuchenden Softwaresystems auf definierte Eingabedaten ausgewertet. Es wird also das Programmverhalten zur Laufzeit anhand bestimmter vorher festgelegter Anwendungsfälle untersucht.
Auf diese Weise können z.B. während der semantischen Analyse entstandene Fragestellungen aufgeklärt werden.

(3) Erfahrung

Trotz aller bewährter Methodik darf eines nicht außer Acht gelassen werden: Erfahrung. Ein so komplexer Prozess wie das Reverse Engineering stellt auch Anforderungen an denjenigen, der diesen ausführt.
Im Reverse Engineering-Prozess wird ein System auf Anhaltspunkte hin analysiert, die zu einer Hypothese über die Semantik bestimmter

[3] Toter Code in Schleifen o.ä. kann so nicht ermittelt werden

Codefragmente führen könnten. Das Erkennen dieser Anhaltspunkte basiert im Wesentlichen auf kognitiven Prozessen, die hauptsächlich durch Heuristiken geleitet werden, die wiederum auf den Erfahrungen des durchführenden Softwareingenieurs aufbauen [Jahnke00]. Durch den Erfahrungsschatz des Ausführenden wird Reverse Engineering immer auch zu einem individuellen Prozess, was letztlich dazu führt, dass der Reverse Engineering-Prozess nicht als allgemeingültiger Vorgang abstrahiert werden kann.

Werkzeugunterstützung

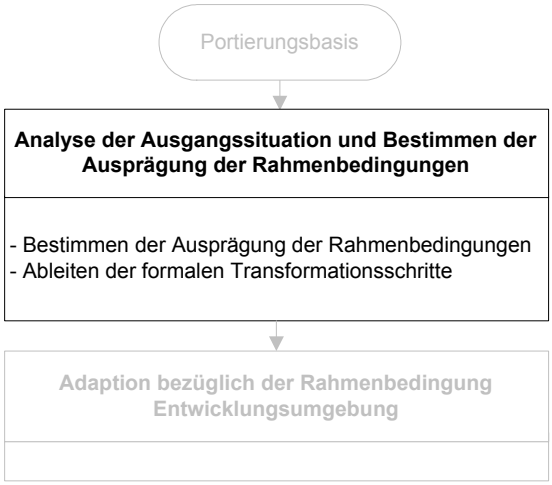
Die im Rahmen des Reverse Engineering angeführten Analysemethoden können alle mit entsprechendem Aufwand manuell durchgeführt werden. Allerdings besteht dann ein gewisses Risiko der Unkorrektheit und Unvollständigkeit der Analyseergebnisse. Auch wächst der Aufwand der Analyse mit der Projektgröße und ist ab einem bestimmten Zeitpunkt nicht mehr zu bewältigen. Aus diesen Gründen existiert eine Reihe von Werkzeugen zur Unterstützung der Abläufe des Reverse Engineering-Prozesses.

Zunächst seien die statischen Analysemethoden betrachtet. Hierbei ist die lexikalische Analyse wohl am besten unterstützt, da sie im Wesentlichen nur Zeichenkettenvergleiche erfordert.
Schwieriger wird es, die Werkzeugunterstützung bei der semantischen Analyse zu realisieren. Für den Parser einer Programmiersprache ist es kein Problem, die Syntax eines Quelltextes zu überprüfen (wie es bei der syntaktischen Analyse der Fall ist). Es ist aber schier unmöglich, den Sinngehalt einer Quelltextpassage automatisiert in menschlicher Sprache ermitteln zu lassen. Daher existieren Werkzeuge, die mittels Syntaxparser vom Quelltext abgeleitete Dokumente erstellen, die dem Anwender ein Gerüst zur weiteren Bearbeitung bieten (siehe dazu Anhang C – Werkzeuge, Doxygen). Die erstellten Dokumente führen jeden Bezeichner, jede Funktion, Klasse etc. in einer übersichtlichen Notation auf. Damit fällt es dem Benutzer leichter, den Quelltext in Hinblick auf seine Semantik systematisch zu analysieren. Diese Analysemethode kommt also nicht ohne manuellen Aufwand aus.
Anders dagegen die Querverweis-Analyse. Hier existieren Werkzeuge, die mit Hilfe eines Syntaxparsers vollständige Querverweislisten eines Quelltextes generieren.

Werkzeugunterstützung ist auch für die dynamische Analyse verfügbar. Hierbei kann z.B. mit Hilfe von Debuggern oder Profilern das Programmverhalten während der Laufzeit untersucht werden. Die Gruppe dieser Werkzeuge ist durch enorme Komplexität gekennzeichnet, da nicht mehr nur innerhalb der Grenzen des zu untersuchenden Programms operiert wird, sondern auch in Teilen der Ausführungsumgebung. Dies ist notwendig, da es möglich sein muss, das zu untersuchende Programm während der Laufzeit zu unterbrechen, um z.B. den Programmzustand nach einer bestimmten Eingabe zu untersuchen.

Eine für diese Arbeit besonders wichtige und eingangs kurz erwähnte Art von Werkzeugen des Reverse Engineering sind die s.g. Decompiler bzw. Disassembler.
Da der Quelltext eines zu untersuchenden Softwaresystems nicht immer vorliegt, der Binärcode aber nicht ohne weiteres durch Menschen lesbar ist, besteht Bedarf an Werkzeugen, die in der Lage sind, diesen Binärcode in eine menschenlesbare Repräsentation zu übertragen. Decompiler erzeugen aus dem Binärcode einen Quelltext der entsprechenden Programmiersprache und Disassembler übersetzen ihn in ein Assemblerprogramm. Natürlich entsprechen die so erzeugten Quellen keineswegs exakt den Quellen, die dem ursprünglichen Erstellungsvorgang zugrunde lagen. Dies liegt darin begründet, dass zum einen Programmiersprachenkonstrukte (z.B. *inline*-Anweisungen in C) und Bezeichner bei der Erstellung des Binärcodes verloren gehen und manche Strukturen im Rahmen von Optimierungen durch andere ersetzt werden.
Bei dem vorliegenden Projekt wurde ein Disassembler verwendet, um eine DLL zu untersuchen, die lediglich als Binärdatei vorliegt.

3. Analyse der Ausgangssituation und Bestimmen der Ausprägungen der Rahmenbedingungen



Dem im vorherigen Kapitel erstellten Vorgehensmodell folgend besteht der erste Schritt der Portierung in der Analyse der Ausgangssituation. Dieser Schritt setzt sich aus drei Unterschritten zusammen, welche Host- und Zielsystem bezüglich der Ausprägung jeweils einer Rahmenbedingung hin untersuchen. Mit ihren Ergebnissen bildet die Analyse der Ausgangssituation die elementare Grundlage für die eigentliche Adaption der Programmquellen.

Analyse Entwicklungsumgebung

Hostsystem

Als Entwicklungsumgebung für *XCTL* wurde *Borland C++* verwendet. Ursprünglich mit der Version *Borland C++ 4.5* begonnen, wurde *XCTL* bereits auf *Borland C++ 5.02* angepasst. *Borland C++* ist im Wesentlichen, abgesehen von einigen proprietären Sprach- und Syntaxweiterungen, ANSI-C++ kompatibel. Für die Erstellung von Dialogen ist die integrierte Entwicklungsumgebung von *Borland C++* mit einem Ressourceneditor und einem Ressourcencompiler ausgestattet. Das Entwicklungssystem stellt eine ganze Reihe eigener Bibliotheksfunktionen zur Verfügung. Umfangreiche Dokumentationen über den Sprach- und Bibliotheksumfang werden im Rahmen der Entwicklungsumgebung bereitgestellt [Borland 97].

Zielsystem

Als Zielentwicklungsumgebung ist *Microsoft Visual C++ 6* vorgesehen. Dabei handelt es sich um eine Entwicklungsumgebung mit 32-bit C++ Compiler, Ressourceneditor und Debugger. Sie ist im Vergleich zu *Borland C++ 5.02* deutlich moderner.
Auch diese Entwicklungsumgebung ist im Wesentlichen ANSI-C++ kompatibel, wobei allerdings auch proprietäre Spracherweiterungen definiert wurden. Für viele Anwendungsfälle der Windowsprogrammierung werden Funktionsbibliotheken zur Verfügung gestellt. Microsoft hat sowohl Sprach- und Bibliotheksumfang von *Visual C++* als auch Änderungen gegenüber Vorgängerversionen von Entwicklungsumgebung und Betriebssystem detailliert dokumentiert [Microsoft 99b].

Schlussfolgerung

Die für die Ableitung portierungsrelevanter Informationen notwendige genaue Analyse der Eigenschaften der beiden Entwicklungsumgebungen ist äußerst umfangreich und schwierig. Zwar gibt es erschöpfende Informationen bezüglich der jeweiligen Entwicklungsumgebung, es ist aber praktisch unmöglich, diese im Rahmen dieses Einzelprojektes gegenüberzustellen, um relevante Unterschiede aufzuzeigen.
Wesentlich ist, dass beim Wechsel vom Hostsystem zum Zielsystem die Programmiersprache nicht gewechselt wird. Die ANSI-C++ Kompatibilität beider Entwicklungsumgebungen sichert zu, dass die Sprachumfänge beider Umgebungen eine ausreichend große Schnittmenge bilden und demzufolge zu konvertierende sprachlichen Elemente einen überschaubaren Umfang haben. Erleichternd kommt hinzu, dass Compiler und Linker der Zielentwicklungsumgebung bei jedem nicht unterstützten Sprachelement, bzw. jedem nicht auflösbaren Bibliotheksaufruf entsprechende Fehler- und Warnmeldungen ausgeben. Aus diesen Gründen ist es als akzeptabel anzusehen, dass auf eine allumfassende Analyse der Unterschiede von Host- und Zielentwicklungsumgebung verzichtet werden kann. Statt dessen dienen die von der Zielentwicklungsumgebung generierten Fehler- und Warnmeldungen als Indikatoren für Problemstellen im Quelltext.
Dieses als *Bottom-Up-Methode der Portierung* bezeichnete Vorgehen [Müller 02] ist besonders im vorliegenden Falle effizienter als eine aufwändige

Recherche der Unterschiede von Host- und Zielumgebung, da das vorliegende Einzelprojekt nicht alle Systemunterschiede tangiert.
Anzumerken ist, dass die Verfügbarkeit einer Aufstellung oder Datenbank mit allen Systemunterschiedenen definitiv von Vorteil für das Ausführen mehrerer Portierungsprojekte wäre.

Analyse Softwareumgebung

Hostsystem

In der Ausgangsversion kann XCTL nur unter dem Betriebssystem Windows 3.11 eingesetzt werden. Dabei handelt es sich jedoch nicht um ein Betriebssystem im eigentlichen Sinne, sondern vielmehr um eine grafische Benutzerschnittstelle, die als Aufsatz für DOS konzipiert wurde. Ebenso wie DOS für sich allein, ist auch die Kombination aus DOS und Windows 3.11 ein 16-bit Betriebssystem mit den entsprechenden Eigenschaften und Einschränkungen, wenngleich auch der Windows 3.11-Aufsatz schon einige fortschrittliche Erweiterungen gegenüber reinem DOS einführt. So existiert mit Windows 3.11 erstmals ein Treiberkonzept, wenn auch noch recht eingeschränkt (nur für Standardgeräte wie Grafikkarten, Mäuse etc.). Zugriffe auf Hardware, die nicht von diesem Treiberkonzept abgedeckt werden, erfolgen direkt mit den programmiersprachenüblichen Befehlen[1]. Weiterhin bietet Windows 3.11 erstmals ein nachrichtenbasiertes System zur einheitlichen Gestaltung von Programmoberflächen, was in Verbindung mit dem ebenfalls unterstützten kooperativen Multitasking eine neue Möglichkeit der Aufteilung von Prozessressourcen unter den Prozessen darstellt. Programme können damit quasi gleichzeitig laufen. Durch das nachrichtenbasierte Fenstersystem und die Kapselung der Grafikhardware durch Treiber ist es möglich, grafische Programmausgaben wesentlich zu vereinfachen und sicherzustellen, dass diese auf verschiedenen Systemen identische Resultate erzielen. Ein weiterer Mechanismus der in Bezug auf die Anforderungen von XCTL zu erwähnen ist, sind DLLs. DLLs fassen bestimmte Funktionen zu Bibliotheken zusammen, die bei Bedarf geladen werden können und somit die Systemressourcen schonen. Der Code einer DLL wird nur einmal geladen. Unter Windows 3.11 teilen sich die verschiedenen Instanzen einer DLL das Datensegment, was bedeutet, alle Instanzen teilen sich dieselben globalen Variablen.

[1] *inp(,) outp()* etc., siehe Tabelle [D.4]

Eine Besonderheit von Windows 3.11 besteht in der Art der Speicherverwaltung. Dabei wird das Modell des *segmentierten Speichers* verwendet. Der physikalische Adressraum wird in 65.536 Segmente zu je 64kByte unterteilt. Innerhalb eines Segments wird ein Byte durch ein 16-bit breites Offsetregister adressiert. Damit kann auf ein Objekt im Speicher über Segment:Offset –Paare zugegriffen werden. Da der 8086 mit seinem 20-bit Adressbus aber nur 1MByte adressieren kann, bei 64k x 64kByte aber theoretisch 4GByte zu adressieren sind, werden die Segmente in einem Abstand von 16 Byte miteinander verzahnt (vergl. [Messmer98]). Diese Verzahnung ist die eigentliche Besonderheit, da Speichersegmentierung auch unter Win32 verwendet wird. Das Modell der Speicherverwaltung unterstützt keinerlei Sicherheitsmechanismen, so dass systemkritische Speicherbereiche nicht geschützt sind und z.B. fehlerhafte Programme das System destabilisieren können. Die Implementierung eines 16-Bit Windowsprogramms muss auf dieses Modell angepasst sein: zum einen in der Art der Funktionsaufrufe, zum anderen dahingehend, dass der sorgfältige Umgang mit dem Speicher in der Verantwortung des Entwicklers liegt.
In Tabelle [3.1] sind die wesentlichen Eigenschaften von Windows 3.11 noch einmal zusammengefasst.

Windows 3.11
- 16-Bit Ausführungsumgebung (basiert auf DOS) - Grafische Nutzeroberfläche (max. 256 Farben/gerastert) - Kooperatives Multitasking - Segmentierte Speicheradressierung - DLL-Instanzen nutzen gemeinsamen Speicher - Hardwarezugriffe direkt möglich - Nur rudimentäres Treiberkonzept vorhanden - Keine Verwaltung von Hardwareressourcen

Tabelle [3.1]

Zielsystem

Als echte 32-bit Betriebssysteme und Windows 3.11-Nachfolger bietet sich die Familie der Windows NT-basierten Betriebssysteme als neue Plattform an. Derzeit sind das die Versionen NT 5.0 (bekannt als Windows 2000) und NT 5.1 (bekannt als Windows XP). Die Betriebssysteme der Familie Windows 9x/Me sind zwar auch als 32-bit Systeme bekannt, jedoch handelt es sich bei ihnen um keine echten 32-bit Systeme, da sie immer noch auf DOS basieren. Sie sind aus softwaretechnischer Sicht weniger weit entwickelt als die NT Familie (Sicherheitskonzept, Ressourcenschutz etc.). Hinzu kommt, dass Microsoft den gesamten Support für diese Systeme eingestellt hat, was sich mittelfristig auch auf die technische Unterstützung von Hardwareproduzenten auswirken wird (z.B. die Bereitstellung von Gerätetreibern).

Zielplattform der Portierung ist Windows 2000. Dieses Betriebssystem verwendet für die grafische Nutzerschnittstelle das gleiche Nachrichtensystem wie Windows 3.11 seinerzeit. Änderungen sind entsprechend dokumentiert (siehe Tabellen [D.2], [D.3], [D.4]). Im Gegensatz zu Windows 3.11 ist bei Windows 2000 die Grafikausgabe mit einer Farbtiefe von bis zu 32-bit möglich.

(1) Windows 2000 implementiert preemptives Multitasking. D.h., im Gegensatz zu kooperativem Multitasking entscheiden nicht die Prozesse selber über die Abgabe der Systemressourcen an andere Tasks, sondern der Taskscheduler des Betriebssystems.

(2) Bei Windows 2000 handelt es sich um ein sehr stabiles System. Es ist so gut wie ausgeschlossen, dass eine Benutzeranwendung das gesamte System zum Absturz bringent. Um diese Stabilität realisieren zu können, existieren zwei Prozessorzugriffsmodi: der *Benutzermodus* und der *Kernelmodus* (siehe dazu Abb. [3.1]).
Benutzeranwendungen werden im *Benutzermodus* ausgeführt, Betriebssystemcode (also auch Gerätetreiber) hingegen im *Kernelmodus*. Der *Kernelmodus* oder auch *privilegierter Modus* ist ein Ausführungsmodus, in dem der Prozessor den vollen Zugriff auf das gesamte System hat. Im Gegensatz dazu ist der Zugriff auf das System im *Benutzermodus* eingeschränkt, das bedeutet, dass nur auf bestimmten Speicher (nämlich nur den zum Prozess gehörenden) zugegriffen werden kann, und dass die

Ausführung bestimmter CPU-Instruktionen[2] nicht erlaubt ist. Durch dieses Prinzip wird sichergestellt, dass ein Fehlverhalten (egal, ob beabsichtigtes oder unbeabsichtigtes) einer Benutzeranwendung nicht die Stabilität des gesamten Systems gefährdet. Allerdings bedeutet dies für eine Benutzeranwendung auch, dass sie nicht mehr direkt die Funktionen bestimmter Hardware, wie Erweiterungskarten, nutzen kann. Diese putative Einschränkung zwingt Benutzeranwendungen dazu, für den Zugriff auf Erweiterungshardware Gerätetreiber zu verwenden. In Windows 2000 ist dafür ein vollständiges Treiberkonzept realisiert, dass an dieser Stelle kurz dargestellt werden soll.

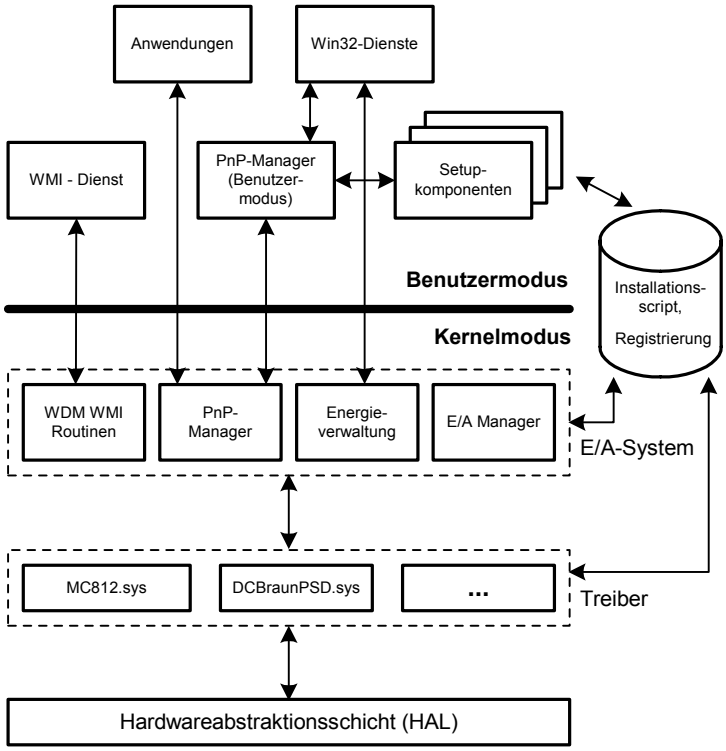


Abbildung [3.1] zeigt die Komponenten von Windows 2000 und ihren Bezug zu den Ausführungsmodi *Benutzermodus* und *Kernelmodus*. Quelle: [Solomon00].

Einen Teil des komplexen E/A-Systems (vergl. Abb. [3.1]) von Windows 2000 bilden die Gerätetreiber. Sie stellen eine E/A-Schnittstelle für einen bestimmten Gerätetyp zur Verfügung, mit Hilfe derer dann, über weitere Zwischeninstanzen, eine Anwendung auch aus dem Benutzermodus auf die Hardware zugreifen kann. Die wesentlichen Teile des E/A-Systems sind der *E/A-Manager* und der *PnP-Manager*. Die *Windows Management Instrumentation (WMI)* sowie die *Energieverwaltung* sollen hier nur der

[2] unter anderem die Maschinenbefehle *in, out*

Vollständigkeit halber erwähnt sein, da sie im später folgenden Portierungsprozess keine Rolle spielen. Der *E/A-Manager* bildet die Grundlage für die Kommunikation zwischen Anwendungen und Gerätetreiber. Anforderungen an einen Gerätetreiber (oder an eine andere E/A–Systemkomponente) werden in E/A–Anforderungspakete , oder auch *IRP*s (I/O Request Packets) genannt, verpackt. Es handelt sich dabei um eine Datenstruktur, welche die E/A-Anforderung vollständig beschreibt. Der *E/A-Manager* ermittelt den entsprechenden Gerätetreiber und übermittelt das Paket an diesen, welcher dann entsprechend der Anforderung handeln kann und seinerseits das Paket an den *E/A-Manager* zurück übergibt, mit dem Resultat der Anforderung. Über den *E/A-Manager* ist es ebenfalls für einen Treiber möglich, Anforderungen an andere Treiber zu senden.

Windows 2000 Gerätetreibertypen		
Typ		**Kurzbeschreibung**
Dateisystemtreiber		setzen an Dateien gerichtete E/A Anforderungen um in entsprechende Anforderungen an Treiber von Massenspeichern, Netzwerkgeräten
Legacy Treiber		Treiber, die für Windows NT 4.0 geschrieben wurden, unter Windows 2000 mit Einschränkungen verwendbar, da keine PnP-Managerunterstützung existiert
Bildschirmtreiber		Übersetzung von geräteunabhängigen Grafikanforderungen in geräteabhängige
WDM Treiber		Windows Driver Model, beinhaltet Unterstützung für PnP-Manager und Energieveraltung
	Bustreiber	verwalten logischer oder physischer Bussysteme
	Funktionstreiber	verwalten bestimmten Gerätetyp, exportieren operationale Schnittstelle des Geräts in Betriebssystem
	Filtertreiber	ergänzen das Verhalten eines Geräts
Virtuelle Gerätetreiber		dienen der Emulation von 16-bit MS-DOS Anwendungen
Druckertreiber		Übersetzung von geräteunabhängigen Grafikanforderungen in druckerspezifische Befehle
Geräteklassentreiber		implementieren E/A-Verarbeitung einer bestimmten Geräteklasse
Anschlusstreiber		implementieren E/A-Verarbeitung von Anforderungen die an einen bestimmten Anschlusstyp gerichtet sind
Miniporttreiber		ordnen generische E/A-Anforderung, die an einen bestimmten Anschlusstyp gerichtet sind, einem bestimmten Adaptertyp zu

Tabelle [3.2], in Anlehnung an Quelle [Solomon00]

Der *PnP-Manager* ist die Komponente des E/A- Systems, welche die Verwaltung der Hardwareressourcen übernimmt. Er sorgt unter anderem dafür, dass der für ein PnP – Gerät passende Treiber geladen wird und

weisst jedem Treiber, der Hardwareressourcen benötigt um ein ihm zugeordnetes Gerät zu verwalten, entsprechende Hardwareressourcen zu. Durch diese zentrale Verwaltungsinstanz im System können Hardwareressourcenkonflikte vermieden werden.
Tabelle [3.2] enthält eine Übersicht der unter Windows 2000 möglichen Gerätetreibertypen.

(3) Eine weitere Neuerung ist die lineare Speicherverwaltung. Windows 2000 kann einer Anwendung Speicher aus einem 2 GByte großen Bereich zuweisen. Auf diesen, für die Anwendung exklusiven, Speicherbereich kann die Applikation linear zugreifen. Die Zeiger für den Zugriff auf den Speicher sind 32-bit breit.

(4) Als letzte herausragende Eigenschaft von Windows 2000 sei das Konzept der Benutzerverwaltung erwähnt. Mit diesem können sicherheitsrelevante Objekte (z.B. Dateien) je nach Bedarf nur einem bestimmten Nutzerkreis zugänglich gemacht werden. Damit lässt sich der Zugriff auf den Computer ganz oder teilweise einschränken.
Tabelle [3.3] fasst die wesentlichen Eigenschaften von Windows 2000 übersichtlich zusammen.

Eigenschaften von Windows 2000
- Echte 32-Bit Ausführungsumgebung (kein unterliegendes DOS)
- Grafische Nutzeroberfläche (bis 32-Bit Farbtiefe)
- Preemptives Multitasking
- Lineare Speicheradressierung
- Speicherschutz/DLL - Instanzen nutzen getrennten Speicher
- Hardwarezugriffe nicht direkt möglich (nur für privilegierte Software/Treiber)
- Zentrale Verwaltung der Hardwareressourcen
- Nutzerprofile mit Rechteverwaltung

Tabelle [3.3]

In [Microsoft 99b] sind Bibliotheksfunktionen und Datenstrukturen beschrieben, die beim Zielsystem Windows 2000 als Nachfolger von Windows 3.11 nicht mehr unterstützt werden. Auch Veränderungen an Funktionen oder Datenstrukturen sind dort aufgeführt ebenso wie Funktionen, die zwar immer noch unterstützt werden, aber überholt sind.

Die von Microsoft angegebenen offiziellen Mindestanforderungen an die PC-Hardware für den Einsatz von Windows 2000 sind in Tabelle [3.4] zusammengefasst, Quelle: [Microsoft99a].

Hardwaremindestanforderungen für Windows 2000
- min. Intel Pentium 133 MHz, bzw. vergleichbarer - min. 32 MByte Hauptspeicher (64MB empfohlen, mehr verbessert die Perfomance) - Festplattenkapazität durchschnittlich 2 GB mit 650MB freiem Speicherplatz - VGA Bildschirm - Tastatur, Maus - CD-ROM Laufwerk für die Installation

Tabelle [3.4]

Schlussfolgerung

Aus den zusammengetragenen Eigenschaften von Host- und Zielsystem können folgende Schlussfolgerungen gezogen werden:

(1) Nachrichtensystem
Aufgrund des von beiden Systemen verwendeten Nachrichtensystems für die Fensterverwaltung kann diese bei der Portierung in weiten Teilen unverändert übernommen werden. Änderungen an der Fensterverwaltung beschränken sich auf die unter Anhang D Tabelle [D.2.2] aufgeführten Unterschiede zwischen beiden Systemen.

(2) Multitasking
Beide Systeme unterstützen Mutitasking, allerdings in verschiedenen Varianten. Das von Windows 2000 unterstützte preemptive Multitasking stellt insofern einen entscheidenden Vorteil dar, als dass ein Task, der z.B. auf das Ergebnis einer I/O-Operation wartet, nicht mehr das gesamte System blockieren kann (wie unter Windows 3.11 möglich). Ebenfalls ist es damit möglich, ein Programm in einzelne Threads aufzuteilen und somit innerhalb des Programms bestimmte Blockaden zu vermeiden[3]. Diese Argumente sind jedoch weniger portierungsrelevant im eigentlichen Sinne, als vielmehr Hinweise für eine der Portierung möglicherweise folgende Erweiterung bzw. Optimierung des XCTL-Systems., deshalb sei in diesem Zusammenhang auf Kapitel 9 *Ausblick* verwiesen.

[3] z.B. E/A bedingte Blockaden, so das die Oberfläche nicht reagiert wenn Hardwarezugriffe stattfinden

(3) Speicherverwaltung
Dieser Bereich gehört mit zu den wichtigsten Unteschieden, da Host- und Zielbetriebssystem ist in diesem Punkt gravierend differieren. Die Vereinfachung des Speicherzugriffes, den die lineare Speicherverwaltung mit sich bringt, wirkt sich auch auf das zu portierende System aus. Sämtliche komplizierte Berechnungen von Segment:Offset Adresspaaren müssen entfernt werden. NEAR, FAR oder ähnliche Modifizierer können entfallen. Alle Zeiger sind fortan 32-bit breit.

(4) Benutzerrechte
Durch die unter Windows 2000 neu hinzugekommenen Möglichkeiten der Benutzerrechteverwaltung bieten sich auch hier Optimierungen für zukünftige Versionen an (siehe Kapitel 9 Ausblick). Portierungsrelevante Informationen ergeben sich hieraus jedoch nicht.

(5) Ressourcenverwaltung/Speicherschutz
Durch die Umsetzung einer zentralen Ressourcenverwaltung und eines restriktiven Speicherschutzes sind direkte Zugriffe auf die Hardware und auf nicht zur Applikation gehörenden Speicher unter dem Zielbetriebssystem Windows 2000 nicht mehr möglich. Damit ist bei der Analyse der Quellen von XCTL besonders auf die Verwendung entsprechender Hardware- / Speicherzugriffsmechanismen zu achten. Da solche Zugriffe nicht mehr erlaubt sind, muss eine adäquate Alternative für den Zugriff auf die Hardware gefunden werden. Dieser Punkt ist für die Portierung von besonders hoher Relevanz, da es sich bei XCTL um ein Hardwaresteuerungssystem handelt.

(6) Die unter [Lauer 92], [Microsoft 99b] aufgeführten Hinweise auf Änderungen an Bibliotheksfunktionen und Datenstrukturen können kommentarlos in den Pool der portierungsrelevanten Informationen aufgenommen werden.

Analyse Hardwareumgebung

Die im XCTL-System eingesetzte Hardware kann grob in zwei Kategorien eingeteilt werden:

(1) anwendungsspezifische Hardware (Motorsteuerkarten mit Präzisionsmotoren und Detektorsteuerkarten mit Röntgen-detektoren)

(2) Hardware der Ausführungsumgebung (Steuerungs-PCs)

Die anwendungsspezifische Hardware bleibt unverändert, Host- und Zielsystem sind identisch. Die Hardware der Ausführungsumgebung des Zielsystems hingegen muss sich nach den Bedürfnissen von Zielbetriebssystem und anwendungsspezifischer Hardware richten. In Anhang F ist beschrieben, wie die Hardware der Ausführungsumgebung unter den beschriebenen Voraussetzungen ausgewählt wurde.

Anwendungsspezifische Hardware – Hostsystem, Zielsystem[4]

Gegenwärtig wird an den Messplätzen die in den Tabellen [3.5.1] und [3.5.2] aufgeführte anwendungsspezifische Hardware genutzt.

anwendungsspezifische Hardware: Motorcontroller			
MotorContoller	**Bus-Interface**	**Kommunikation**	**Anzahl steuerbarer Motoren**
C-812	8-bit ISA	Memory-mapped-I/O	4
C-832	16-bit ISA	Port-I/O (opt. IRQ)	2

Tabelle [3.5.1]

anwendungsspezifische Hardware: Detectorcontroller		
DetectorController	**Bus-Interface**	**Kommunikation**
ASA-Karte (BraunPSD)	8-bit ISA	Port-I/O
Radicon	8-bit ISA	Port-I/O
AX5216 *)	8-bit ISA	Port-I/O

Tabelle [3.5.2]

*) bei dieser Karte handelt es sich nicht um eine Detektorsteuerkarte im eigentlichen Sinne, sondern um eine generische Zählerkarte, die dafür verwendet wird, die durch ein externes Strahlenmessgerät (Typ RFT 20046) bereitgestellten Zählimpulse einzulesen.

Bei diesen ISA-Erweiterungskarten handelt es sich um Geräte, die einen eigenen Prozessor besitzen. Damit wird eine autonome Steuerung der externen Geräte (Motoren, Detektoren) durch die Karte erreicht. Der PC wiederum ist in der Lage, durch Festlegen von Parametern und Senden

[4] Quellen: [Axiom 92], [Braun 94], [Radicon 95], [PI 95]

von Befehlen die auf der Karte laufenden Prozesse zu beeinflussen und Daten zu ermitteln. Um diese Kommunikation mit den Karten realisieren zu können, verwendet die anwendungsspezifische Hardware drei Mechanismen. Diese sind in Tabelle [3.6] aufgeführt.

Mechanismen für die Kommunikation mit der Hardware		
Typ	**Eigenschaften**	**Programmierbeispiel**

Port-I/O	- Kommunikation über speziellen I/O-Bus
	- Register der Hardware werden an Ports/Adressen dieses Busses gebunden
	- 16-Bit Adressen, 8 Bit Datenbreite
	- Lesen und Schreiben
		`//lesen`
		`int Value =`
		`inp(Port);`
		
		`//schreiben`
		`outp(Port, Value);`

Memory-mapped-I/O	- Kommunikation über 'normale' Speicheradressen
	- Register der Hardware werden auf Speicheradressen abgebildet
	- 16-Bit Adressen (Segment:Offset)
	- Lesen und Schreiben
		`char* Register =`
		`0xD000;`
		
		`//lesen`
		`char Value =`
		`*Register;`
		
		`//schreiben`
		`*Register = Value;`

IEEE 488	- externer Bus - 8-Bit Datenbreite - ähnlich LPT-Schnittstelle	-Programmierung über spezielle DLL (WIN488.DLL)

Tabelle [3.6]

Zu den Mechanismen im Einzelnen:

(1) Port-I/O
Bei diesem Kommunikationsmechanismus wird ein I/O-Port über eine 16-bit Adresse auf dem I/O-Bus angesprochen. Ein I/O-Port ist 8-bit breit und kann prinzipiell gelesen und geschrieben werden. Dies ist jedoch auch von der Hardware abhängig, die auf den adressierten Port reagiert. Eine Erweiterungskarte kann die I/O-Ports nutzen, indem sie eine bestimmte Anzahl von Ports okkupiert und zum Beispiel Register dort direkt anlegt. Eine entsprechende Semantik vorausgesetzt, sind damit z.B. Befehlsregister, Datenregister und Statusregister realisierbar, die für den PC (und somit die darauf laufende Applikation) für direkten Zugriff zur Verfügung stehen.

(2) Memory-mapped-I/O
Bei diesem Mechanismus wird ein bestimmter Bereich des Hauptspeicheradressraums dem Gerät zugeordnet (gemapped). Ein

Zugriff auf eine Adresse eines solchen Speicherbereichs resultiert in einem Zugriff auf ein Register der Kartenhardware. Dieser Mechanismus bietet sich besonders für Geräte an, die einen hohen Bedarf an I/O-Ports besitzen, aber aufgrund der begrenzten Anzahl von Ports (maximal 65535) diesen Bedarf mittels Port-I/O nicht decken können. Weit verbreitet ist dieser Mechanismus bei Grafikkarten.

(3) GPIB(IEEE488)
Dieser Industriestandard beschreibt einen 8-bit breiten externen Datenbus, der Ähnlichkeiten mit der parallelen Schnittstelle eines PCs aufweist. Er ist in XCTL bislang zwar softwaretechnisch vorgesehen, wurde jedoch in Ermangelung der dafür zusätzlich benötigten Hardware nie verwendet.

Bei den beschriebenen Arten der Kommunikation ist es sehr wahrscheinlich, dass ein externes Gerät mit einer bestimmten Aktion beschäftigt ist und auf Anforderungen durch den PC nicht reagieren kann. In einem solchen Falle hat der PC (bzw. die Anwendung) zwei Möglichkeiten. So kann der Status des betroffenen Gerätes periodisch abgefragt werden, um den Zeitpunkt der Bereitschaft des Gerätes zu ermitteln. Dies ist jedoch ein sehr ressourcenaufwändiger Prozess, da der Prozessor durch das ständige Abfragen andere Aufgaben unterbrechen muss bzw. diese gar nicht wahrnehmen kann. Aus diesem Grunde existiert ein Mechanismus, bei dem die betroffene Erweiterungskarte dem PC selbständig Bereitschaft signalisiert. Über so genannte *IRQ*s (Interrupt ReQuests) kann eine Erweiterungskarte veranlassen, dass eine bestimmte Funktion von der CPU ausgeführt wird (z.B. das Auslesen von Daten oder das Senden neuer Befehle). Der Motorcontroller C-832 unterstützt zwar diesen Mechanismus, gegenwärtig wird er jedoch von XCTL nicht verwendet.

Motorsteuerung

Die Eigenschaften der derzeit verwendeten Motorsteuerkarten sind in Tabelle [3.7] aufgeführt.

Eigenschaften der Motorsteuerkarten	
Typ	**Eigenschaften**
C-812	- programmierbarer Mikrocontroller (herstellerspezifische Firmware in EProm) - Ansteuerung von bis zu 4 Motoren, Positionsrückmeldung und Endlagenschalter - 8-Bit ISA-Schnittstelle - IEEE488-Schnittstelle (zusätzliche Hardware erforderlich) - Memory-mapped-I/O, IEEE 488 - Spannungsversorgung der Motoren intern/extern
C-832	- 2 spezielle, nicht programmierbare Mikrocontroller (ASIC) - Ansteuerung von max. 2 Motoren, Positionsrückmeldung und Endlagenschalter - 16-Bit ISA-Schnittstelle - Port-I/O - Spannungsversorgung der Motoren intern

Tabelle [3.7]

Detektornutzung

Zur Messung radioaktiver Strahlung kommen drei Typen von Detektoren zum Einsatz. Die Detektortypen im Einzelnen:

(1) 0-dimensionale Detektoren
Zu einem Zeitpunkt ermittelt ein solcher Detektor einen einzelnen, positionsunabhängigen Messwert. Die grafische Darstellung des Messergebnisses ist eine Kurve, wobei die Intensität über der Zeit dargestellt wird. Diese Detektoren werden als SCSCS (Single Channel Scintilation Crystal Spectrometer) bezeichnet.

(2) 1-dimensionale Detektoren
Bei 1-dimensionalen Detektoren wird zu einem Zeitpunkt eine Menge von positionsabhängigen Messwerten ermittelt. Je nach Genauigkeit variiert die Menge der möglichen Kanäle, von denen jeder einen Messwert zu einem Zeitpunkt liefert. Solche Detektoren werden auch als PSD (Position Sensitive Detector) bezeichnet. Die grafische Darstellung des Messergebnisses ist eine Kurvenschar (eine Kurve je Kanal).

(3) 2-dimensionale Detektoren
Ein 2-dimensionaler Detektor ist in der Lage positionsabhängige Messwerte einer Fläche zu liefern. Vereinfacht beschrieben, könnte ein

solcher Detektor als Kamera bezeichnet werden, die empfindlich für radioaktive Strahlung ist. Die grafische Darstellung der Messergebnisse ist eine Menge von Flächen (Bilder).

Tabelle [3.8] gibt Auskunft über die Detektortypen und deren Vertreter, sowie die derzeitige Unterstützung dieser durch XCTL.

Detektortypen		
Typ	**Vertreter**	**derzeitige Unterstützung durch XCTL**
0-dimensionale Detektoren	Radicon SCSCS	Ja
	Russischer SCSCS (BDS-6-06) an RFT Strahlenmessgerät (im Folgenden als *Generic SCSCS* bezeichnet)	Ja
1-dimensionale Detektoren	Braun PSD	Ja
	Stoe PSD	nein, da in autarkes Steuerungssystem eingebunden
2-dimensionale Detektoren	Proscan CCD-Kamera, eigener Controller HSSC-1	Nein

Tabelle [3.8]

Hardware der Ausführungsumgebung

Ein Messplatzrechner hat, vereinfacht auf die technischen Anforderungen eines Messplatzes, im Wesentlichen folgende zwei Punkte zu erfüllen: Zum einen müssen Motoren gesteuert werden um Proben, Detektoren positionieren zu können. Zum anderen müssen Detektoren abgefragt werden um Messwerte zu erhalten (siehe Kapitel 1 - Anwendungsdomäne).

Hostsystem

Die in der Physik seit Begin des Einsatzes von XCTL bis zum aktuellen Zeitpunkt verwendete Hardware besteht aus IBM AT – kompatiblen PC-Systemen. Überwiegend handelt es sich dabei um Rechner mit

Zentralprozessoren des Typs 80486DX33 mit 66 MHz Taktfrequenz. Die Systeme haben eine Hauptspeicherausstattung im Bereich von 8 - 32 MB und Festplattenkapazitäten zwischen 100MB und 2GB.

Zielsystem

In Tabelle [3.9] sind die Eigenschaften des zum aktuellen Zeitpunkt verfügbaren Hardwarepools aufgeführt.

Verfügbarer Hardwarepool für das Hardwarezielsystem
- Intel Pentium I MMX 133 –200 Mhz - 128 – 256 MByte Hauptspeicher (PS/2) - Festplattenkapazität ab 2 GB - 3 – 5 ISA-Steckplätze - VGA Bildschirm - Tastatur, Maus - CD-ROM

Tabelle [3.9]

Schlussfolgerung

Da Host- und Zielsystem für den Bereich der anwendungsspezifischen Hardware identisch sind, ergeben sich hieraus keine portierungsrelevanten Informationen.
Der Vergleich der Eigenschaften von Host- und Zielsystem der Hardware der Ausführungsumgebung zeigt, dass keine Unterschiede bestehen, die einen direkten Einfluss auf die Portierung hätten: Es handelt sich in beiden Fällen um Computer mit der gleichen Architektur. Lediglich die Leistung des Zielsystems ist gegenüber der des Hostsystems um ein Vielfaches besser. Prinzipiell wirkt sich bessere Leistung der Hardware im Allgemeinen positiv auf ältere Software aus, da die gesteigerte Verarbeitungsgeschwindigkeit einen Vorteil für die Abarbeitung von Funktionen oder ganzen Prozessen mit sich bringt. Aber genau dies kann auch dazu führen, dass beispielsweise Synchronisationsmechanismen, welche durch systemnahe Warteschleifen realisiert wurden, also nicht zeitabsolut sind, nicht mehr funktionieren. Damit wäre ein indirekter Einfluss des Zielsystems auf die Portierung gegeben, so dass dieser Punkt letztendlich doch als portierungsrelevante Information gewertet werden muss.

Zusammenfassung der Schlussfolgerungen

Die Analyse der Ausgangssituation hat gezeigt, dass das größte Problempotential der Portierung in den Unterschieden der *Softwareumgebungen* steckt. Besonders die Umstände der veränderten Möglichkeiten für den Zugriff auf die Hardware sind dabei hervorzuheben. Ohne vorweg greifen zu wollen, sei bereits hier darauf hingewiesen, dass für den Hardwarezugriff unter Windows 2000 Gerätetreiber realisiert werden müssen. In diesem Zusammenhang ist die Analyse der anwendungsspezifischen Hardware indirekt portierungsrelevant, da nur mit Hilfe der Informationen dieser Analyse entsprechende Gerätetreiber erstellt werden können.
Die Rahmenbedingungen *Entwicklungsumgebung* und *Hardware* sind in ihren Unterschieden zwischen Host- und Zielsystem weniger umfangreich.
Bei der Portierung bezüglich der Zielentwicklungsumgebung handelt es sich nicht um die Portierung von einer auf eine völlig andere Programmiersprache. Größere semantische und syntaktische Adaptionen sind somit nicht zu erwarten. Die Unterschiede der Entwicklungsumgebungen bestehen in Bibliotheksfunktionen und Spracherweiterungen. Dies ist nicht als gravierendes Problem anzusehen, da die Zielentwicklungsumgebung ihrerseits automatisch auf nicht unterstützte Elemente aufmerksam macht.
Bei der Hardware sind bis auf die zu erwartenden Probleme bei hardwarenaher Programmierung durch die Leistungssteigerung des Zielsystems keine weiteren Probleme zu erwarten.
Damit ist eine Informationsbasis geschaffen, die im weiteren Verlauf als Grundlage für Analysen des Quelltextes der XCTL Hostversion dient. Es wird somit möglich den Quelltext zielgerichtet nach Code abzusuchen, der direkt in Bezug zu den ermittelten Ausprägungen der untersuchten Rahmenbedingungen steht. Dieser Code muss dann entsprechend den Gegebenheiten des Zielsystems adaptiert werden.

Aufbauend auf die Theorie der Portierung in Kapitel 2 können folgende formelle Transformationsschritte definiert werden.

(1) Anpassung der Konstrukte der Implementationssprache des Portierungsgegenstandes auf die Anforderungen der Zielumgebung.

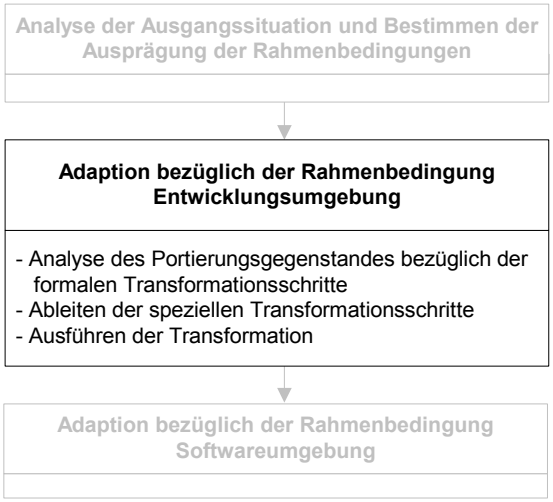
(2) Anpassung der Nutzung von entwicklungsumgebungs-spezifischen Bibliotheksfunktionen.

(3) Transformation der Nutzung der Systemschnittstellen

(4) Kompensation der Auswirkungen interner Funktions-mechanismen

Diese Transformationen werden in den folgenden Kapiteln aufgegriffen und präzisiert und in ihrer Anwendung dargestellt.

4. Analyse und Adaption bezüglich der Rahmenbedingung Entwicklungsumgebung



Dieses Kapitel beschäftigt sich mit den bezüglich der Rahmenbedingung *Entwicklungsumgebung* notwendigen anpassungen von *XCTL*. Die in Kapitel 2 erläuterten grundlegenden Adaptionen dieser Portierungsart werden hier am Beispiel von *XCTL* praktisch nachvollzogen.

Ziel dieses Schrittes ist die Überführung der Quell- und Projektdateien von *Borland C 4.5x/5.x* nach *Microsoft Visual C++ 6.0*. Das Ergebnis ist dann eine mit *Visual C++* erstellte, lauffähige Win32-Applikation mit teilweise noch Win16-spezifischen Quelltextelementen. Bedingt durch die Unmöglichkeit, die speziellen Hardwarekomponenten zu nutzen, verfügt die Anwendung noch nicht über ihre wesentliche Funktionalität.
Für die Durchführung dieses Schrittes bilden die während der Analyse der Ausgangssituation bezüglich der Rahmenbedingung *Entwicklungsumgebung* gewonnenen Informationen die Grundlage. Diese Informationen dienen dazu, den Quelltext zielgerichtet nach Problemstellen mit Bezug zur Rahmenbedingung Entwicklungsumgebung zu untersuchen.
Basis der hier erörterten Anpassungen ist die von Dragan Bojic erstellte Anleitung der Portierung von *XCTL* nach *Visual C++* [Bojic 02]. Da sich der Portierungsgegenstand seither in nicht unerheblichem Maße verändert hat, dienen seine Ausführungen lediglich der Orientierung.

Notwendige Transformastionsschritte

Zusammenfassend werden die in Kapitel 3 erarbeiteten Transformationsschrittte hier nochmals kurz erläutert.

(1) Anpassung der Konstrukte der Implementationssprache des Portierungsgegenstandes auf die Anforderungen der Zielumgebung.

Da sich die Syntax einiger Sprachelemente und zwischen Host- und Zielsystem zwar nur geringfügig, aber dennoch unterscheidet, ist eine Anpassung der entsprechenden Quelltexttteile notwendig.

(2) Anpassung der Nutzung von entwicklungsumgebungsspezifischen Bibliotheksfunktionen.

Einige der genutzten Bibliotheksfunktionen und vordefinierten Datenstrukturen des Hostsystems sind im Zielsystem nicht mehr verfügbar. Für die entsprechenden Quelltextteile ist demnach eine Anpassung auf Funktionen und Datenstrukturen des Zielsystems erforderlich.

Ein vorbereitender Schritt, der nicht in der Portierungstheorie erläutert wurde, besteht aus der Erstellung einer adäquaten Projektumgebung auf dem Zielsystem.

Prinzipiell muss der Adaptierung des Quelltextes eine Analyse desselben vorausgehen. Dabei werden die durch die Analyse der Rahmenbedingungen ermittelten Informationen dazu verwendet, mittels statischer bzw. dynamischer Analyse die Codeteile zu lokalisieren, die adaptiert werden müssen, um volle Kompatibilität mit dem Zielsystem zu gewährleisten. Wie in Kapitel 3 (Analyse Entwicklungsumgebung) beschrieben ist, existiert keine allumfassende Informationsbasis bezüglich der Rahmenbedingung *Entwicklungsumgebung*. Statt dessen wird die *Bottom-Up-Methode der Portierung* unter Verwendung der Entwicklungsumgebung selbst angewendet. Der Schritt der Analyse entfällt damit nicht etwa, sondern verschmilzt quasi mit der Adaption. Zusätzlich werden auch die o.g. formellen Transformationsschritte miteinander kombiniert. Damit lassen sich folgende Schritte für die Vorgehensweise entsprechend Analyse und Adaption bezüglich der Rahmenbedingung *Entwicklungsumgebung* festlegen:

1) Anlegen eines neuen Visual C++ Projektes und Konfiguration der Projekteinstellungen
   (stellt ein Äquivalent zum Borland C – Projekt dar)

2) Bottom-Up-Methode („try and error“) mittels Compiler und Linker der Zielentwicklungsumgebung:
   a) Übersetzen der Quellen und Analysieren der eventuell gemeldeten Fehler und Warnungen mit Ermittlung der betroffenen Codeteile
   b) Adaption der ermittelten Codeteile
   c) Wiederholung von a) und b), bis keine Fehlermeldungen mehr auftreten

Anlegen eines neuen Visual C++ Projektes und Konfiguration der Projekteinstellungen

Der erste Schritt der Portierung umfasst die allgemeine Einrichtung eines neuen Projekts in der Zielumgebung. Dieses Projekt wird dann in weiteren Schritten adäquat konfiguriert und angepasst. Ziel ist es, die ursprüngliche Projektstruktur möglichst genau nachzubilden.

(1) Anlegen eines Arbeitsbereiches *XControl.dsw*.
(2) Anlegen eines Visual C++ Projektes je Modul.

anzulegende Visual C++ Projekte		
Modulname	**Projektname**	**Ziel**
XControl.exe	XControl	Win32 Anwendung
SPLib.dll	SPLib	Win32 Dynamic Link Library
Counters.dll	Counters	Win32 Dynamic Link Library
Motors.dll	Motors	Win32 Dynamic Link Library
Hwio.dll	Hwio	Win32 Dynamic Link Library
Protocol.dll	Protocol	Win32 Dynamic Link Library

(3) Festlegen der Modulabhängigkeiten
Dies beeinflusst die Erstellungsreihenfolge der einzelnen Module und ist elementar für die korrekte Auflösung der exportierten/importierten Symbole. Generell ist ein Modul von einem anderen abhängig, sobald es Symbole nutzt/importiert,

welche durch das andere Modul definiert/exportiert werden.

festzulegende Modulabhängigkeiten	
Modul	**Abhängig von**
XControl	Counters, Motors, Protocol, SPLib
SPLib	-
Counters	Hwio, Motors, SPLib
Motors	Hwio, SPLib
Hwio	-
Protocol	SPLib

(4) Zuordnung der einzelnen Quelldateien in der Verzeichnisstruktur zum jeweils entsprechenden Projekt. Grundlage hierfür ist Zuordnung im Ausgangssystem.[1]

Zuordnung der Quelldateien	
Modul	**Quellcodedateien**
XControl	m_arscan.cpp, m_data.cpp, m_device.cpp, m_dlg.cpp, m_dlgdif.cpp, m_justag.cpp, m_main.cpp, m_scan.cpp, m_steerg.cpp, m_topo.cpp, matrix.cpp, mespara.cpp, mesparaw.cpp, mj_funk.cpp, mj_gui.cpp, mj_ofunk.cpp, mj_ogui.cpp, mj_old.cpp, msimstat.cpp, tp_funk.cpp, tp_gui.cpp, transfrm.cpp
SPLib	dlg_tpl.cpp, l_layer.cpp, m_curve.cpp, u_files.cpp, u_timer.cpp, u_values.cpp, splib.def
Counters	detec_hw.cpp, detecgui.cpp, detecmes.cpp, detecuse.cpp, dllmain.cpp, counters.def
Motors	m_layer.cpp, motors.cpp, motors.def
Hwio	hwio.cpp, hwio.def
Protocol	mespara.cpp, p_lang.cpp, prdiprnw.cpp, procombw.cpp, proprnw.cpp, protdif.cpp, protdifw.cpp, proto.cpp, protocol.cpp, protow.cpp, protparw.cpp, protpasw.cpp, prottop.cpp, prottopw.cpp, prtoprnw.cpp, protocol.def

Zuordnung der Headerdateien	
Modul	**Headerdateien**
XControl	autojust.h, difrkmty.h, m_arscan.h, m_data.h, m_dlg.h, m_dlgdif.h, m_justag.h, m_scan.h, m_steerg.h, matrix.h, mespara.h, mesparaw.h, mj_funk.h, mj_gui.h, mj_ofunk.h, mj_ogui.h, mj_old.h, msimstat.h, topogrfy.h, tp_funk.h, tp_gui.h, transfrm.h, workflow.h
SPLib	datavisa.h, evrythng.h, l_layer.h, swintrac.h, u_files.h, u_timer.h, u_utils.h, u_values.h
Counters	detec_hw.h, detecgui.h, detecmes.h, *detectorcontroller.h*,

[1] Die im Rahmen der Portierung neu erstellten Dateien sind kursiv dargestellt.

	detecuse.h, range.h
Motors	c_8x2.h, ieee.h, m_layer.h, m_motcom.h, m_mothw.h, *motorcontroller.h*, motrstrg.h, msimstat.h
Hwio	hwguids.h, hwio.h
Protocol	p_lang.h, prdipara.h, prdiprn.h, procombw.h, proprnw.h, protdif.h, protdifw.h, proto.h, protocol.h, protow.h, protparw.h, protpasw.h, prottop.h, prottopw.h, prtopara.h, prtoprnw.h

Zuordnung der Ressourcendateien	
Modul	**Ressourcendateien**
XControl	applica.ico, circlecr.cur, main.rc
SPLib	splib.rc
Counters	counters.rc
Motors	motors.rc
Hwio	-
Protocol	protocol.rc

(5) Konfiguration der Projekteinstellungen der Module.
Hier werden nur die Konfigurationsparameter dargestellt, welche von den jeweiligen Standardwerten abweichen. Eine Erläuterung der Konfigurationsparameter folgt weiter unten.

Konfiguration der Projekteinstellungen - XControl			
Ort	**Parameter**	**Wert (Debug)**	**Wert (Release)**
C/C++	Run-Time-Type-Informationen (RTTI) aktivieren	aktiviert	aktiviert
	Zusätzliche Include-Verzeichnisse	..\..\INCLUDE\	..\..\INCLUDE\
	Projekt-Optionen	- Löschen von /MTd bzw. MLd - Setzen von MDd	- Löschen von /MTd bzw. MLd - Setzen von MDd
	Präprozessor-Definitionen	Use_Counters, Use_Motors, Use_Library, Use_Protocol	Use_Counters, Use_Motors, Use_Library, Use_Protocol
Linker	Objekt-/Bibliothek-Module	winmm.lib, splib.lib, motors.lib, counters.lib, protocol.lib	winmm.lib, splib.lib, motors.lib, counters.lib, protocol.lib
	Zusätzlicher Bibliothekpfad	..\..\OBJ\Debug	..\..\OBJ\Release
Bearbeitung nach dem Erstellen	Befehl(e) nach dem Erstellen	copy .\Debug\XControl.exe ..\..\Exe\Debug	copy .\Debug\XControl.exe ..\..\Exe\Release

Konfiguration der Projekteinstellungen - SPLib			
Ort	**Parameter**	**Wert (Debug)**	**Wert (Release)**
C/C++	Run-Time-Type-Informationen (RTTI) aktivieren	aktiviert	aktiviert
	Zusätzliche Include-Verzeichnisse	..\..\INCLUDE\	..\..\INCLUDE\
	Projekt-Optionen	- Löschen von /MTd bzw. MLd - Setzen von MDd	- Löschen von /MTd bzw. MLd - Setzen von MDd
	Präprozessor-Definitionen	Build_Library	Build_Library
Linker	Objekt-/Bibliothek-Module	winmm.lib	winmm.lib
	Zusätzlicher Bibliothekpfad	..\..\OBJ\Debug	..\..\OBJ\Release
Bearbeitung nach dem Erstellen	Befehl(e) nach dem Erstellen	copy .\Debug\SPLib.dll ..\..\Exe\Debug	copy .\Debug\SPLib.dll ..\..\Exe\Release

Konfiguration der Projekteinstellungen - Counters			
Ort	**Parameter**	**Wert (Debug)**	**Wert (Release)**
C/C++	Run-Time-Type-Informationen (RTTI) aktivieren	aktiviert	aktiviert
	Zusätzliche Include-Verzeichnisse	..\..\INCLUDE\	..\..\INCLUDE\
	Projekt-Optionen	- Löschen von /MTd bzw. MLd - Setzen von MDd	- Löschen von /MTd bzw. MLd - Setzen von MDd
	Präprozessor-Definitionen	Build_Counters, Use_Library, Use_Hardware	Build_Counters, Use_Library, Use_Hardware
Linker	Objekt-/Bibliothek-Module	winmm.lib, splib.lib, motors.lib, hwio.lib	winmm.lib, splib.lib, motors.lib, hwio.lib
	Zusätzlicher Bibliothekpfad	..\..\OBJ\Debug	..\..\OBJ\Release
Bearbeitung nach dem Erstellen	Befehl(e) nach dem Erstellen	copy .\Debug\Counters.dll ..\..\Exe\Debug	copy .\Debug\Counters.dll ..\..\Exe\Release

Konfiguration der Projekteinstellungen - Motors			
Ort	**Parameter**	**Wert (Debug)**	**Wert (Release)**
C/C++	Run-Time-Type-Informationen (RTTI) aktivieren	aktiviert	aktiviert
	Zusätzliche Include-	..\..\INCLUDE\	..\..\INCLUDE\

	Verzeichnisse		
	Projekt-Optionen	- Löschen von /MTd bzw. MLd - Setzen von MDd	- Löschen von /MTd bzw. MLd - Setzen von MDd
	Präprozessor-Definitionen	Build_Motors, Use_Library, Use_Hardware	Build_Motors, Use_Library, Use_Hardware
Linker	Objekt-/Bibliothek-Module	winmm.lib, splib.lib, hwio.lib	winmm.lib, splib.lib, hwio.lib
	Zusätzlicher Bibliothekpfad	..\..\OBJ\Debug	..\..\OBJ\Release
Bearbeitung nach dem Erstellen	Befehl(e) nach dem Erstellen	copy .\Debug\Motors.dll ..\..\Exe\Debug	copy .\Debug\Motors.dll ..\..\Exe\Release

Konfiguration der Projekteinstellungen - Hwio			
Ort	**Parameter**	**Wert (Debug)**	**Wert (Release)**
C/C++	Run-Time-Type-Informationen (RTTI) aktivieren	aktiviert	aktiviert
	Zusätzliche Include-Verzeichnisse	..\..\INCLUDE\	..\..\INCLUDE\
	Projekt-Optionen	- Löschen von /MTd bzw. MLd - Setzen von MDd	- Löschen von /MTd bzw. MLd - Setzen von MDd
	Präprozessor-Definitionen	Build_Hardware, Use_Motors, Use_Counters	Build_Hardware, Use_Motors, Use_Counters
Linker	Objekt-/Bibliothek-Module	setupapi.lib, motors.lib, counters.lib	setupapi.lib, motors.lib, counters.lib
	Zusätzlicher Bibliothekpfad	..\..\OBJ\Debug	..\..\OBJ\Release
Bearbeitung nach dem Erstellen	Befehl(e) nach dem Erstellen	copy .\Debug\Hwio.dll ..\..\Exe\Debug	copy .\Debug\Hwio.dll ..\..\Exe\Release

Konfiguration der Projekteinstellungen - Protocol			
Ort	**Parameter**	**Wert (Debug)**	**Wert (Release)**
C/C++	Run-Time-Type-Informationen (RTTI) aktivieren	aktiviert	aktiviert
	Zusätzliche Include-Verzeichnisse	..\..\INCLUDE\	..\..\INCLUDE\
	Projekt-Optionen	- Löschen von /MTd bzw. MLd - Setzen von MDd	- Löschen von /MTd bzw. MLd - Setzen von MDd
	Präprozessor-	Build_Protocol,	Build_Protocol,

	Definitionen	Use_Library	Use_Library
Linker	Objekt-/Bibliothek-Module	winmm.lib, splib.lib	winmm.lib, splib.lib
	Zusätzlicher Bibliothekpfad	..\..\OBJ\Debug	..\..\OBJ\Release
Bearbeitung nach dem Erstellen	Befehl(e) nach dem Erstellen	copy .\Debug\Protocol.dll ..\..\Exe\Debug	copy .\Debug\Protocol.dll ..\..\Exe\Release

Erläuterung der Konfigurationsparameter

- Run-Time-Type-Information (RTTI) aktivieren:

  RTTI erlaubt es, den Typ eines Objekts zur Laufzeit des Programms zu ermitteln. Das Aktivieren obiger Option veranlasst den Compiler entsprechenden Programmcode in die Objektdatei einzufügen und ist Voraussetzung für das einwandfreie Funktionieren typabhängiger Konvertierungsoperationen (z.B. *dynamic_cast*).

- Zusätzliche Include-Verzeichnisse

  Das korrekte Setzen dieses Parameters ist Voraussetzung für den Erfolg des Erstellungsvorganges, da sonst notwendige Headerdateien nicht gefunden werden. Die Notwendigkeit der Konfiguration dieses Parameters ergibt sich zwangsweise aus der Verzeichnisstruktur des Gesamtprojektes.

- Projekt-Optionen Setzen von /MD und Löschen von /MT bzw. /ML

  Diese Änderung dient der Umstellung des Erstellungsziels auf Multithreading.

- Präprozessor-Definitionen

  Die Definition der folgenden Symbole ermöglicht die Nutzung ein und der selben Headerdatei sowohl für den Export als auch den Import von Funktionen und Daten über Modulgrenzen hinweg.

  - *Build_xxx*: Definiert bestimmte Funktionen und Klassen des Moduls xxx als *__declspec(dllexport)* und ermöglicht so den Export dieser Symbole. Dieses Symbol wird in dem Modul definiert, welches die entsprechende Funktionalität zur Verfügung stellt.
  - *Use_xxx*: Definiert bestimmte Funktionen und Klassen des Moduls xxx als *__declspec(dllimport)* und ermöglicht so den

Import dieser Symbole. Dieses Symbol wird in dem Modul definiert, welches die entsprechende Funktionalität nutzt.

- xxx steht dabei für: Library, Hardware, Motors, Counters, Protocol

Objekt-/Bibliothek-Module	
splib.lib	Stubs der im Modul SPLib definierten Funktionen
counters.lib	Stubs der im Modul Counters definierten Funktionen
motors.lib	Stubs der im Modul Motors definierten Funktionen
hwio.lib	Stubs der im Modul Hwio definierten Funktionen
protocol.lib	Stubs der im Modul Protocol definierten Funktionen
winmm.lib	Ein Einbinden dieser Bibliothek beseitigt einige Linker-Warnungen bzgl. unaufgelöster externer Symbole (z.B. *timeSetEvent())*
setupapi.lib	Ein Einbinden dieser Bibliothek ermöglicht die Verwendung von *SetupDixxx*-Funktionen, welche zur Verwaltung von Gerätetreibern benötigt werden.

- Zusätzlicher Bibliothek-Pfad

  Das korrekte Setzen dieses Parameters ist Voraussetzung für den Erfolg des Bindungsvorganges, da sonst notwendige Bibliotheken nicht gefunden werden. Die Notwendigkeit der Konfiguration dieses Parameters ergibt sich zwangsweise aus der Verzeichnisstruktur des Gesamtprojektes.

- Bearbeitung nach dem Erstellen

  Die oben angegebene Kommandozeile kopiert die jeweiligen Erstellungsprodukte in ein gemeinsames Ausgabeverzeichnis. Alle zur Ausführung des Programms benötigten Dateien werden somit in einem Verzeichnis zusammengefasst.

Bottom-Up-Methode der Portierung

Bei der Buttom-Up-Methode werden vom unveränderten Quelltext ausgehend die zu adaptierenden Codeteile mit Hilfe von Linker und Compiler der Zielentwicklungsumgebung ermittelt. Da bei nicht unterstützten Sprach- oder Bibliothekselementen durch Compiler bzw. Linker entsprechende Fehler- oder Warnmeldungen generiert werden, wird der betreffende Code quasi automatisch ermittelt. Die Adaption eines solchen Codeteils kann dann unter Umständen auf andere Codeteile mit dem gleichen Problemelement übertragen werden. Zur Unterstützung

der Anpassung kann auf die umfangreiche Dokumentation der Entwicklungsumgebungen zurückgegriffen werden.
Nach erfolgter Adaption wird ein neuer Übersetzungsversuch gestartet. Wenn keine Fehler mehr auftreten, ist das Ziel dieses Portierungsschrittes erreicht. Ist dies nicht der Fall, so muss dieser Vorgang wiederholt werden, bis keine Fehler mehr auftreten.

Da die durchgeführten Adaptionen nur schwer in einem einheitlichen Schema dargestellt werden können, sind nicht alle der nachfolgenden Tabellen gleich aufgebaut. Sie enthalten aber alle nötigen Informationen, um die dargestellten Anpassungen im entsprechenden Kontext der Quellen nachvollziehen zu können.

(1) Anpassen von Headerdateien

Entfernen von #include <dir.h> und #include <except.h>	
Borland C++	Bedeutung unklar
Visual C++	Diese Dateien existieren nicht und werden auch nicht benötigt

(2) Anpassen der DLL Eintrittsfunktionen

Umbennen von *BOOL DllEntryPoint(…)* bzw. *LibMain(...)* in *BOOL WINAPI DllMain(...)*	
DLL-Eintrittsfunktion (Laden/Entladen der DLL)	
Borland C++	DllEntryPoint(Win32), LibMain(Win16)
Visual C++	DllMain()

(3) Anpassen vordefinierter Symbole und Konstanten

Umbennenen des Speicherklassen-Attributs *_export* nach *__declspec(dllexport)*	
Der Einsatz dieses Symbols ermöglicht den Export von derart definierten Funktionen, Daten und Objekten durch den Linker. Die Definition des DLL-Interfaces durch eine entsprechende .DEF-Datei kann entfallen.	
Borland C++	_export
Visual C++	__declspec(dllexport) (muss vor dem Schlüsselwort WINAPI bzw. CALLBACK stehen)

Umbenennen des Speicherklassen-Attributs *_import* nach *__declspec(dllimport)*	
Der Einsatz dieses Symbols ermöglicht den Import von derart definierten Funktionen, Daten und Objekten aus anderen Modulen.	
Borland C++	_import
Visual C++	__declspec(dllimport) (muss vor dem Schlüsselwort WINAPI bzw. CALLBACK stehen)

Löschen des Symbols *__rtti*	
Aktivieren von Runtime-Type-Information zur korrekten Auflösung von virtual-Klassenmembern	
Borland C++	__rtti
Visual C++	Dieses Symbol gibt es nicht, statt dessen Aktivieren von RTTI in den Projekteinstellungen

Umbenennen von Konstanten	
-	
Borland C++	MAXDIR (max. Länge des Verzeichnisanteils eines Pfades) MAXFILE (max. Länge eines Dateinamens) MAXPATH (max. Länge eines kompletten Pfades) MAXEXT (max. Länge eines Dateierweiterungsstrings) MAXDRIVE (max. Länge eines Laufwerkbezeichners)
Visual C++	_MAX_DIR _MAX_FNAME _MAX_PATH _MAX_EXT _MAX_DRIVE

Definieren des Symbols *M_PI*	
Konstante π	
Borland C++	vordefiniertes Symbol
Visual C++	Da es dieses Symbol nicht gibt, muss es selbst definiert werden #define M_PI 3.14159265358979323846 in internls\evrythng.h

Ändern des Versionsmakros von *__BORLANDC__* nach *_MSC_VER*	
Liefert die Compilerversion	
Borland C++	__BORLANDC__
Visual C++	_MSC_VER

(4) Anpassen von Precompiler-Direktiven

Entfernen von *#pragma argused* bzw. *#pragma argsused*	
Borland C++	Unterbindet die Warnung, dass Funktionsparameter innerhalb der

	Funktion nicht verwendet werden
Visual C++	dieses Pragma existiert nicht, der gewünschte Effekt kann durch Anpassen des Warnungslevels erreicht werden

Anpassen von *#pragma warn*	
Unterbindet die in der Direktive spezifizierte Compiler-Warnung	
Borland C++	#pragma warn #Warnung
Visual C++	#pragma warning (disable : #Warnung)

Entfernen von *#pragma message*	
Borland C++	
Visual C++	syntaktisch falsch, Äquivalent unbekannt

Entfernen von *#pragma code_page*	
Borland C++	
Visual C++	syntaktisch falsch, Äquivalent unbekannt

Umbenennen von *WORKSHOP_INVOKED* in *RC_INVOKED*	
Vordefiniertes Symbol, welches definiert ist, wenn der ResourceCompiler läuft und ermöglicht so compilerabhängige Übersetzung. (In den .rc-Dateien kann #ifndef WORKSHOP_INVOKED komplett entfallen.)	
Borland C++	WORKSHOP_INVOKED
Visual C++	RC_INVOKED

(5) Anpassen von Datentypen

Umbenennen von *struct time* und *struct date* in *SYSTEMTIME* und Umbenennen der entsprechenden Elemente.	
Datenstruktur für Zeit und Datum mit jeweils eigenen Elementen für jede Komponente (SYSTEMTIME ersetzt struct time und struct date). Nach der Umbenennung treten ggf. semantisch unnütze doppelte Variablendeklarationen auf, die entfernt werden können.	
Borland C++	typedef struct time { … } typedef struct date { … }
Visual C++	typedef struct _SYSTEMTIME { … } SYSTEMTIME;

Entfernen von *typedef unsigned long UINT32*	
Datentyp unsigned Integer (32 Bit), zu entfernen in porting\types.h	
Borland C++	nutzerdefinierter Datentyp
Visual C++	in basetsd.h schon äquivalent vordefiniert

Umbenennen des Datentyps *HMETAFILE* nach *HENHMETAFILE*	
Handle für ein Windows-Metafile	
Borland C++	HMETAFILE
Visual C++	HENHMETAFILE

(6) Anpassen des Aufrufs Borland-spezifischer Bibliotheksfunktionen

Umbenennen von *fnsplit* nach *_splitpath*	
Funktion zur Zerlegung eines Pfades in seine Komponenten	
Borland C++	fnsplit()
Visual C++	_splitpath()

Umbenennen von *fnmerge* nach *_makepath*	
Funktion zur Erstellung eines Pfades aus einzelnen Komponenten	
Borland C++	fnmerge()
Visual C++	_makepath()

Umbenennen von *chdir* nach *_chdir*	
Funktion zum Setzen des aktuellen Laufwerks	
Borland C++	chdir()
Visual C++	_chdir() + Einbinden des Headerfiles <direct.h> in m_arscan.cpp, m_scan.cpp, l_layer.cpp, m_main.cpp

Umbenennen von *gettime()*, *getdate()* in *GetLocalTime()*	
Ermittelt das aktuelle lokale Datum und die aktuelle lokale Zeit. Es treten nach der Umbenennung ggf. semantisch unnütze doppelte Aufrufe von GetLocalTime() auf, die entfernt werden können.	
Borland C++	gettime(), getdate()
Visual C++	GetLocalTime()

Umbenennen von *setdisk* in *_chdrive* , Addieren von 1 zum Parameter	
Setzt das aktuelle Laufwerk	
Borland C++	setdisk(0=A:,1=B:,...)
Visual C++	_chdrive(1=A:,2=B:,...)

Umbenennen von *GetCurrentTime()* in *GetCurTime()*	
Borland C++	Funktionsdefinition, nicht vorbelegt
Visual C++	vordefiniertes Makro für 16-Bit-Applikationen

(7) Bereinigung von Fehlern

Abgleich der Deklaration und Definition von Funktionen	
Problem	speziell *__declspec(xxx)*)
An vielen Stellen stimmen Funktionsdeklaration (.h) und -definition (.cpp) bzgl. des Speicherklassen-Attributs nicht überein.	
Borland C++	_export/_import nur in Funktionsdefinition
Visual C++	__declspec(dllexport/dllimport) sowohl in Funktionsdefinition als auch Deklaration notwendig (sonst Fehlermeldung wegen Neudefinition mit unterschiedlicher Bindung)

Variable Entfernen	
Variable	*HANDLE hTheModule;*
Die schon in jedem Module existierende DLL-globale Variable *hModuleInstance* (*HINSTANCE*) nimmt deren Platz ein. Die Initialisierung der Variablen wird in DllMain() vorgenommen, indem die Zeile *hModuleInstance* = *hModule* im *DLL_PROCESS_ATTACH*-Zweig hinzugefügt wird. Instanzhandle der DLL, Initialisiert in *DllMain()*	
Borland C++	Typ *HANDLE*
Visual C++	Typ *HINSTANCE*

Einfügen	
global	*extern HINSTANCE hModuleInstance;*
Dateien	protocol\prdiprnw.cpp, protocol\protpasw.cpp, protocol\prtopprnw.cpp, protocol\protparw.cpp, motrstrg\motors.cpp, detecuse\detecgui.cpp
Borland C++	-
Visual C++	Die Variable *hModuleInstance* wird jeweils in *protocol\protocol.cpp*, *motrstrg\m_layer.cpp* und *detecuse\dllmain.cpp* modulglobal definiert und muss in anderen nutzenden Quelldateien explizit bekannt gegeben werden (wurde in den o. a. Dateien vorher nicht genutzt).

Entfernen von *WINAPI* bzw. *CALLBACK*	
Durchzuführen in allen Zeilen mit *typedef* und in Deklarationen von Methoden.	
Borland C++	-
Visual C++	syntaktisch falsch

Ersetzen von *try*-Konstrukten	
von	try { var=new..; A;} catch (xalloc){B;}
durch	var = new...; if (!var){ B; } else { A; }
Borland C++	Exception *xalloc* meldet Fehler bei Speicherallokierung
Visual C++	*xalloc* gibt es nicht

Entfernen von *__declspec(dllexport)*	
Position	jeweils vor *_MAXLENCHARACTERISTIC, _MAXLENUNIT, _MAXLENFORMAT*

Datei	motrstrg\m_motcom.h
Borland C++	-
Visual C++	syntaktisch falsch: Export statischer Bezeichner

Ändern von Typkonvertierungen	
von	(FARPROC) lpfnSetPort = GetProcAddress(AsaDllInstance, "SetPort");
in	lpfnSetPort = (TSetPort)GetProcAddress(AsaDllInstance, "SetPort");
Datei	motrstrg\motors.cpp
Borland C++	-
Visual C++	syntaktisch falsch : *GetProcAddress()* liefert *FARPROC*-Zeiger, also muss dieser auf den Typ der Funktion konvertiert werden und nicht umgekehrt.

Ändern der Deklaration von Funktionssignaturen	
von	void (WINAPI *lpfnSetPort) (WORD, WORD); BYTE (WINAPI *lpfnGetPort) (WORD); void (WINAPI *lpfnSetTimeout) (DWORD); int (WINAPI *lpfnGetData) (WORD, WORD, WORD, LPLONG, WORD, long&);
in	typedef void (WINAPI *TSetPort) (WORD, WORD); typedef BYTE (WINAPI *TGetPort) (WORD); typedef void (WINAPI *TSetTimeout) (DWORD); typedef int (WINAPI *TGetData) (WORD,WORD,WORD,LPLONG,WORD, long&);
Datei	detecuse\detecmes.cpp
Borland C++	-
Visual C++	syntaktisch falsch

Ersetzen	
von	CountersDLLInstance = GetModuleHandle("device32.dll");
durch	CountersDLLInstance = GetModuleHandle("counters.dll");
Datei	internls\m_main.cpp
Borland C++	-
Visual C++	Das entsprechende Modul heißt counters.dll

Ersetzen	
von	FileLocations = new TFileLocations(_argv[0]);
durch	char _arg[_MAX_PATH]; GetModuleFileName(hInstance,_arg,_MAX_PATH); FileLocations = new TFileLocations(_arg);
Datei	internls\m_main.cpp (TMain::TMain)
Borland C++	Das Symbol *_argv[]* ist scheinbar vordefiniert
Visual C++	Das Symbol *_argv[]* ist nicht definiert, der Name des Moduls muss also auf alternativem Weg ermittelt werden.

Ersetzen	
von	cmd.P1 = (TCParam)atof(p1);
durch	cmd.P1 = (TCParam)atoi(p1);
Datei	workflow\m_steerg.cpp
Borland C++	-
Visual C++	TCParam ist eine Aufzählung (enum) -> Typ int und nicht float

Zeile ändern	
Hinzufügen eines Semikolons am Ende der Zeile mit dem Label "*SearchAgain*:"	
Datei	internls\l_layer.cpp
Borland C++	-
Visual C++	syntaktisch falsch

Ersetzen	
von	pd.hInstance = (HANDLE) NULL;
durch	pd.hInstance = (HINSTANCE) NULL;
	protocol\proprnw.cpp(TProtocolManagePrintDlg::InitializePrint())
Borland C++	-
Visual C++	HINSTANCE ist in BORLAND und VC unterschiedlich definiert

Ändern der Header-Symboldefinition	
von	#ifndef __MJ_GUI.H #define __MJ_GUI.H
in	#ifndef __MJ_GUI_H #define __MJ_GUI_H
Dateien	manjust\mj_gui.h, manjust\mj_funk.h, manjust\mj_ogui.h, manjust\mj_ofunk.h, manjust\mj_old.h
Borland C++	-
Visual C++	in derart definierten Symbolen ist das Zeichen '.' nicht erlaubt

Ändern der Zeilen	
von	#if (!defined(__BORLANDC__)) // std::string #include <string> using namespace std; ...
in	#if (!defined(__BORLANDC__)) // std::string #include <string> typedef std::string string; …
Datei	u_utils\u_utils.h
Borland C++	-
Visual C++	Namespace-Konflikt zwischen <string> und <fstream.h>, der durch die using namespace Anweisung ausgelöst wird.

Ändern der Zeile	
von	int TStoePsd::PsdRead(int, int, int, LPWORD)
in	int TStoePsd::PsdRead(int, LPWORD)
Diese Zeile befindet sich in einem *#ifdef _WIN32* ... Konstrukt	

Datei	detecuse\detecmes.cpp
Borland C++	-
Visual C++	syntaktisch falsch

Nach Durchführung dieser Modifikationen lassen sich alle Quelldateien kompilieren. Für ein erfolgreiches Linken der Module und die Erstellung der Applikation sind aber noch weitere Modifikationen notwendig.

Vorbereitend werden folgende Quelltexterweiterungen vollzogen, welche für die Portierung nicht zwingend notwendig sind, aber den Portierungsvorgang erheblich erleichtern. Ein Teil dieser Erweiterungen wurden von vorherigen Autoren in den Quellen bereits begonnen, allerdings nicht konsequent angewendet und beendet.

Im Folgenden werden mopdulspezifische Symbole definiert die den Export/Import von Funktionen steuern. Sämtliche Vorkommen von *__declspec(dllexport)* sind in dem jeweiligen Modul durch das entsprechende Symbol zu ersetzen. Durch diese Definitionen wird es möglich, die Headerdateien des Moduls sowohl für den Export als auch den Import der Symbole zu nutzen, in Abhängigkeit von den entsprechenden Direktiven *Build_Xxx* bzw. *Use_Xxx*.

allgemeine Definition	`#if defined (x)`
	`#define y __declspec(dllexport)`
	`#elif defined(z)`
	`#define y __declspec(dllimport)`
	`#else`
	`#define y`
	#endif		

Modul	x	y	z
SpLib.dll	Buil_Library	_CURVECLASS	Use_Library
Motors.dll	Build_Motors	_MOTORCLASS	Use_Motors
Counters.dll	Build_Counters	_COUNTERCLASS	Use_Counters
Hwio.dll	Build_Hardware	_HWIOCLASS	Use_Hardware
Protocol.dll	Build_Protocol	_PROTOCOLCLASS	Use_Protocol

Die folgenden Modifikationen beziehen ihre Motivation direkt aus Fehlermeldungen des Linkers und sind entsprechend grundlegend für den Erfolg der Portierung.

Änderungen aufgrund von Linkerproblemen		
Datei	**Änderung**	**Grund für Änderung**
protocol\	Klasse *TProtocolParameterDlg*	Symbol ist sonst nicht in nutzenden

protparw.h	über *_PROTOCOLCLASS* exportieren	Modulen verfügbar
swintrac\ swintrac.h	Klassen *TModalDlg* und *TModelessDlg* über *_CURVECLASS* exportieren	Symbole sind sonst nicht in nutzenden Modulen verfügbar
internls\ SPLib.def	Hinzufügen von *EXPORTS lpDBase*	Symbol ist sonst nicht in nutzenden Modulen verfügbar
motrstrg\ Motors.def	Hinzufügen von *EXPORTS lpMList*	Symbol ist sonst nicht in nutzenden Modulen verfügbar

Abschließend müssen noch die Ressourcenvorlagen der Nutzerschnittstelle konvertiert werden. Dies ist notwendig, da die mit dem *Borland Resource Workshop* erstellten Dateien nicht kompatibel zum *Resource Compiler* von Microsoft Visual Studio sind. Da sich in diesem konkreten Fall die notwendigen Konvertierungsschritte lediglich auf reine Textersetzungen von Schlüsselwörtern beschränken, kann dieser Portierungsschritt automatisiert erfolgen. Hierzu wird das von Günther Reinecker Jun. entwickelte Werkzeug *RC_Converter* eingesetzt[2]. Ergebnis der Konvertierung sind Visual-Studio-kompatible Ressourcenvorlagen, die bzgl. der Anordnung der Oberflächenelemente manuell nachbearbeitet werden müssen (z.B. überlagern sich Elemente teilweise, Ausrichtungen stimmen nicht).

Als Ergebnis der bisherigen Portierungsschritte lassen sich nun alle Module kompilieren und linken und die Applikation kann in ihrer Gesamtheit als ausführbares Programm erstellt werden.

(8) Bereinigung von Warnungen

Zu diesem Zeitpunkt liegt nun eine Version von XCTL vor, die sich fehlerfrei mit Microsoft Visual C++ erstellen lässt. Letzte offensichtliche Makel sind während des Erstellungsprozesses auftretende Compilerwarnungen, die augenscheinlich durch die strengere Syntaxprüfung des Microsoft - Compilers verursacht werden.

Ersetzen	
von	von struct TDSettings{...}
durch	struct _COUNTERCLASS TDSettings{...}
Dateien	detecuse\detecuse.h
Datentyp wird exportiert und kann in anderen Modulen genutzt werden	

[2] siehe dazu Anhang C – Werkzeuge: RC_Converter.exe

Anpassen	
von	Klasse *CPoint* als *struct* deklarieren
Dateien	detecuse\detecgui.h
CPoint ist hier als *class* deklariert (forward declaration), wird allerdings vorher schon in *datavisa\datavisa.h* als *struct* deklariert	

Anpassen	
von	Exportieren der Klassen *TBasicDialog*, *TBasicWindow* und *TBasicMDIWindow* über *_CURVECLASS*
Dateien	swintrac\swintrac.h
Wenn eine abgeleitete Klasse über *__declspec(dllexport)* exportiert wird, so müssen auch alle ihre Basisklassen auf diese Art und Weise exportiert werden. Die o.g. Klassen sind Basisklassen von *TModalDlg* bzw. *TModelessDlg* und müssen entsprechend exportiert werden.	

Allgemein ergibt sich die Problematik des Exportierens von Basisklassen über *__declspec(dllexport)*, sofern abgeleitete Klassen auf diese Art und Weise exportiert werden. Solange sich die Deklaration/Definition der Basisklasse unter Kontrolle des Entwicklers befindet, stellt dies kein weiteres Problem dar. Schwierig wird es, wenn die Basisklasse jedoch lediglich in Binärform (z.B. als Teil einer Bibliothek) zur Verfügung steht und der Entwickler entsprechend keine Änderungen an der Deklaration/ Definition der Klasse mehr vornehmen kann. In diesem Fall kann die entsprechende Warnung *C4152* nicht korrekt bereinigt werden und es bleibt nur die Möglichkeit diese Warnung zu ignorieren. Dass man dies unter dem beschriebenen Umstand gefahrlos tun kann, ergibt sich aus der trivialen Erkenntnis, dass eine Basisklasse, die Teil einer Bibliothek ist, definitivs als Symbol exportiert wird, wenn auch nicht über *__declspec(dllexport)*.

Ersetzen	
von	std::vector<THotKey*> m_HotKeys;
durch	#pragma warning(push) #pragma warning(disable : 4251) std::vector<THotKey*> m_HotKeys; #pragma warning(pop)
Dateien	swintrac\swintrac.h
Ignorieren der Warnung, dass für std::vector keine Dll-Schnittstelle vorliegt.	

Ersetzen	
von	TRange<DWORD> CountBounds; TRange<float> TimeBounds;
durch	#pragma warning(push)

	#pragma warning(disable : 4251) TRange<DWORD> CountBounds; TRange<float> TimeBounds; #pragma warning(pop)
Dateien	detecuse\detecuse.h
Ignorieren der Warnung, dass für std::vector keine Dll-Schnittstelle vorliegt.	

Ersetzen	
von	implizite Typkonvertierungen in Zuweisungen mit Datenwertverkürzung
durch	explizite Typkonvertierungen ersetzen
gilt allgemein	

Ergänzen	
von	Konstanten
mit	Typbezeichnern (z.B. *1.0f*, für Typ *float*)
gilt allgemein	

(9) Ergänzende Quelltextänderungen

Ersetzen	
von	*WIN32, __WIN32__*
durch	*_WIN32*
WIN32, __WIN32__ sind keine definierten Direktiven, semantisch ist *_WIN32* gemeint	

Ersetzen	
von	String "Borland C++ compiler"
durch	String "Microsoft Visual C++"
Datei	internls\m_main.cpp

Entfernen von Schlüsselworten	
Schlüsselworte	STUB, CODE, DATA, EXETYPE, HEAPSIZE
Dateien	*.DEF
Borland C++	unterstützt
Visual C++	nicht unterstützt

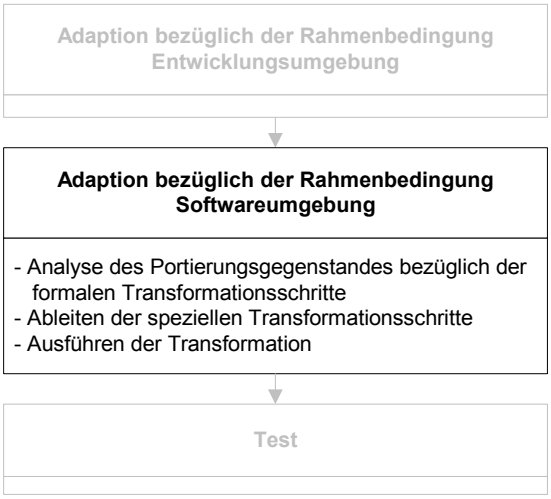
Beschreibung des Transformationsergebnisses

Am Ende dieses Schrittes existiert nun eine XCTL-Version, die bezüglich der Rahmenbedingung *Entwicklungsumgebung* vollständig an das Zielsystem angepasst ist. Das bedeutet, XCTL wird ohne jegliche Fehlermeldung oder Warnung übersetzt. Diese übersetzte Version bietet allerdings nur einen Bruchteil der Funktionalität, welche die XCTL-

Hostversion bereitstellt. So lassen sich beispielsweise sämtliche nicht im Modul *XControl* definierten Dialoge nicht anzeigen. Auch ist kein Zugriff auf die Hardware (Motoren, Detektoren) möglich. Derartige Versuche werden vom Zielbetriebssystem Windows 2000 mit einer Schutzverletzung unterbunden.
Trotz dieser erwarteten Mängel ist nun die Grundlage geschaffen, *XCTL* in einer modernen Entwicklungsumgebung unter Windows 2000 weiter zu portieren bzw. zu entwickeln.

Im nachfolgenden Schritt sind sämtliche noch vorhandene Win16 spezifischen Codeteile durch adäquate Alternativen zu ersetzen, so dass die volle Funktionalität von *XCTL* wieder hergestellt wird.

5. Analyse und Adaption bezüglich der Rahmenbedingung Softwareumgebung



Dieses Kapitel beschäftigt sich mit den bezüglich der Rahmenbedingung *Softwareumgebung* notwendigen Anpassungen von XCTL. Die in Kapitel 2 erläuterten grundlegenden Adaptionen dieser Portierungsart werden hier am Beispiel von XCTL praktisch nachvollzogen.

Notwendige Transformationsschritte

Zusammenfassend werden die in Kapitel 3 erarbeiteten Transformationsschritte hier nochmals kurz erläutert.

(1) Transformation der Nutzung der Systemschnittstellen

Da sich die Syntax der Nutzung der Systemsschnittstellen zwischen Host- und Zielsystem teilweise unterscheidet, ist eine Anpassung der entsprechenden Funktionsaufrufe und damit verbundener Datenstrukturen im Portierungsgegenstand erforderlich.

(2) Kompensation der Auswirkungen interner Funktionsmechanismen

Die im Zielsystem implementierten Mechanismen der Speichertrennung und der Verhinderung direkter Hardwarezugriffe für

Anwendungsprogramme wirken sich negativ auf die Funktionalität des Portierungsgegenstandes aus und erfordern entsprechend umfangreiche Korrekturmaßnahmen.

Transformation der Nutzung von Systemschnittstellen

Unter Win32 (Windows 2000) können die meisten der von Win16 (Windows 3.11) zur Verfügung gestellten und bekannten Systemfunktionen (API-Funktionen) unverändert genutzt werden. Sowohl Aufrufsyntax als auch Funktionalität sind weitestgehend gleich geblieben. Gründe hierfür liegen in der evolutionären Entwicklung der Win32-Betriebssystemfamilie und der vom Hersteller proklamierten Abwärtskompatibilität der Windows-Betriebssysteme. Trotzdem wird die Nutzung einiger dieser Systemfunktionen von Microsoft nicht mehr empfohlen (als „obsolete“ bezeichnet[1]), da sie durch neuere Win32-Funktionen ersetzt wurden oder aufgrund von Änderungen der Systemarchitektur gänzlich entfallen. Diese Funktionsaufrufe im Portierungsgegenstand müssen entsprechend angepasst werden. Hierzu ist es meist ausreichend, den entsprechenden Funktionsaufruf zu ersetzen. Einige Funktionen erfordern allerdings zusätzlich die Modifikation des entsprechenden Aufrufkontextes (z.B. *EnumMetaFile*), was einen höheren Adaptionsaufwand verursacht. Das für Systemfunktionen Gesagte gilt äquivalent für Datenstrukturen und vordefinierte Symbole.
Eine umfassende Zusammenstellung der nicht mehr zu nutzenden Systemelemente ist leider nicht verfügbar. Anhaltspunkte bezüglich notwendiger Adaptionen bieten Compilerfehler und –warnungen sowie Hinweise in der Entwicklerdokumentation für Microsoft Windows [MSDN 03a] und in einschlägiger Literatur (siehe Anhang D, Tab. [D.2] [D.3] [D.4], [Microsoft 99b]). Wurde eine Auflistung potentiell zu ändernder Elemente zusammengetragen, so können entsprechende Quelltextfragemente leicht durch statisches Reverse Engineerung (Suchen im Quelltext) identifiziert werden.

Im Folgenden werden die in XCTL genutzten veralteten Elemente der Systemschnittstellen mit ihren entsprechenden Ersetzungen aufgeführt.

[1] The … function is obsolete. …This function is provided only for compatibility with 16-bit versions of Windows. Win32-based applications should not use this function.

Datenstrukturen

Win16	struct METARECORD
	{
	DWORD rdSize;
	WORD rdFunction;
	WORD rdParm[1];
	}

Win32	struct ENHMETARECORD
	{
	DWORD iType;
	DWORD nSize;
	DWORD dParm[1];
	}

Win16	HMETAFILE
Win32	HENHMETAFILE

Vordefinierte Symbole

Win16	BS_USERBUTTON
Win32	BS_OWNERDRAW

Win16	MB_ICONQUESTION
Win32	MB_ICONEXCLAMATION

Win16	MB_ICONASTERISK
Win32	MB_ICONINFORMATION

Win16	MB_ICONHAND
Win32	MB_ICONSTOP

Systemfunktionen (API)

Win16	FARPROC MakeProcInstance(FARPROC, HINSTANCE)
Win32	Entfällt

Win16	void FreeProcInstance(FARPROC)
Win32	Entfällt

Win16	BOOL FreeModule(HINSTANCE)

Win32	BOOL FreeLibrary(HMODULE)

Win16	BOOL FreeResource(HGLOBAL)
Win32	Entfällt

Win16	HICON LoadIcon(HINSTANCE, LPCSTR)
Win32	HANDLE LoadImage(HINSTANCE, LPCSTR, UINT, int, int, UINT)

Win16	HCURSOR LoadCursor(HINSTANCE, LPCSTR)
Win32	HANDLE LoadImage(HINSTANCE, LPCSTR, UINT, int, int, UINT)

Win16	HDC CreateMetaFile(LPCSTR)
Win32	HDC CreateEnhMetaFile(HDC, LPCTSTR, CONST RECT, LPCTSTR)

Win16	HMETAFILE GetMetaFile(LPCSTR)
Win32	HENHMETAFILE GetEnhMetaFile(LPCTSTR)

Win16	int CALLBACK EnumMetaFileProc (HDC, HANDLETABLE FAR*, METARECORD FAR*, int, LPARAM)
Win32	int CALLBACK EnhMetaFileProc (HDC, HANDLETABLE FAR*, ENHMETARECORD FAR*, int, LPARAM)

Win16	BOOL EnumMetaFile(HDC, HLOCAL, MFENUMPROC, LPARAM)
Win32	HENHMETAFILE EnumEnhMetaFile (HDC, LPCTSTR, HENHMETAFILE, ENHMFENUMPROC, LPVOID, CONST RECT)

Win16	HMETAFILE CloseMetaFile(HDC)
Win32	HENHMETAFILE CloseEnhMetaFile(HDC)

Win16	DWORD GlobalCompact(DWORD)
Win32	entfällt

Win16	HGLOBAL GlobalAlloc(UINT, DWORD)
Win32	LPVOID HeapAlloc(HANDLE, DOWRD, DWORD)

Win16	HGLOBAL GlobalReAlloc(HGLOBAL, DWORD, UINT)
Win32	BOOL HeapReAlloc(HANDLE, DWORD, LPVOID, DWORD)

Win16	HGLOBAL GlobalFree(HGLOBAL)
Win32	BOOL HeapFree(HANDLE, DWORD, LPVOID)

Win16	void FAR* GlobalLock(HGLOBAL)
Win32	Entfällt

Win16	BOOL GlobalUnLock(HGLOBAL)
Win32	Entfällt

Win16	DWORD GlobalSize(HGLOBAL)
Win32	DWORD HeapSize(HANDLE, DWORD, LPCVOID)

Win16	DWORD MoveTo(HDC, int, int)
Win32	BOOL MoveToEx(HDC, int , int , LPPOINT(=NULL))

Systemnachrichten

Die Parameter von WM_COMMAND haben einen anderen Aufbau, siehe dazu [Microsoft 99b] und Tabelle [D.2.2]. Ursächlich dafür ist der Umstand, dass unter Win32 alle Handles 32-bit breit sind und damit einen Parameter allein belegen. Nachrichtenbehandlungsroutinen müssen entsprechend angepasst werden.

Sonstiges

Für die korrekte Funktion von XCTL ist es notwendig sicherzustellen, dass zu jedem Zeitpunkt nur eine einzige Instanz ausgeführt wird. Unter Win16 konnte hierfür eine vom System zur Verfügung gestellte Variable (*hPrevInstance*-Parameter der Funktion *WinMain()*) ausgewertet werden. Unter Win32 ist diese Variable stets mit ihrem Standardwert belegt, so

dass hier ein anderer Mechanismus gefunden werden muss (z.B. Setzen einer systemweiten Mutexvariable).

Hinzufügen	
von	*CreateMutex(NULL,true,"XCTL_32_INSTANCE");* *if (GetLastError() == ERROR_ALREADY_EXISTS)* *{ // only single instance allowed* *exit(0);* *}*
Dateien	internls\m_main.cpp (WinMain() – erste Anweisung)
Dieses Codefragment stellt sicher, dass zu jedem Zeitpunkt nur eine Instanz des Programms ausgeführt wird.	

Weiterhin ist die in den Win16-spezifischen Programmpfaden implementierte Funktionalität so anzupassen, dass diese auch nach Wegfall der entsprechenden Direktive fehlerfrei abgearbeitet werden kann. Alle *_WIN32* Direktiven und damit verknüpfte Programmpfade sind aufzulösen.

Damit sind die wesentlichen Adaptionen dieses Portierungsschrittes vollzogen. Im Weiteren müssen die negativen Auswirkungen interner Funktionsmechanismen von Windows 2000 auf XCTL ausgeglichen werden.

Einschränkungen durch interne Funktionsmechanismen

Speichertrennung

Mit Win32 wurde ein Mechanismus eingeführt, der die Speicherbereiche verschiedener Prozesse effektiv voneinander trennt und diese Trennung auch überwacht. Auf diese Art wurde die Systemstabilität erhöht, da ein fehlerhafter Prozess nun nicht mehr das ganze System negativ beeinflussen kann. Eine Schattenseite dieser Systemeigenschaft ist, dass globale Variablen in DLLs damit auch nicht mehr so einfach nutzenden Prozessen zur Verfügung stehen. Vielmehr müssen diese Variablen explizit exportiert werden. Geschieht dies nicht, kommt es zu Programmabstürzen durch eine s.g. allgemeine Schutzverletzung (General Protection Fault - GPF). Die folgenden Anpassungen beheben diesen portierungsbedingten Fehler.

Ersetzen	
Von	*extern LPDataBase lpDBase;*
Durch	*__declspec(dllimport) LPDataBase lpDBase;*
Dateien	difrkmty\m_arscan.cpp, difrkmty\m_dlgdif.cpp, datavisa\m_data.cpp
Symbol ist sonst nicht verfügbar (GPF)	

Ersetzen	
Von	*extern LPMList lpMList;*
Durch	*__declspec(dllimport) LPMList lpMList;*
Dateien	manjust\mj_funk.cpp, topogrfy\tp_funk.cpp
Symbol ist sonst nicht verfügbar (GPF)	

Hinzufügen	
Von	*extern TMain Main;*
Dateien	swintrac\m_device.cpp, difrkmty\m_dlgdif.cpp, autojust\m_justag.cpp, topogrfy\m_topo.cpp, topogrfy\tp_gui.cpp, mespara\mesparaw.cpp, manjust\mj_gui.cpp, manjust\mj_ogui.cpp, manjust\mj_old.cpp
Die Variable *Main* wird in *internls\m_main.cpp* global definiert und muss in anderen nutzenden Quelldateien explizit bekannt gegeben werden (wurde in den o. a. Dateien vorher nicht genutzt).	

Die entsprechenden Quelltextfragmente wurden durch eine vorangegangene semantische Teilanalyse des Portierungsgegenstandes identifiziert.

Ein weiterer Nebeneffekt der oben beschriebenen Systemeigenschaft zeigt sich beim Zugriff auf die Ressourcen der Nutzeroberfläche. Aufgrund der Speichertrennung sind die modulspezifischen Dialogressourcen von XCTL in getrennten Instanzen definiert und können nur lokalisiert werden, wenn das jeweils entsprechende modulspezifische Instanzhandle bekannt ist. Durch die folgenden Modifikationen wird das jeweilige Instanzhandle der Dialogklasse bekannt gemacht und in den entsprechenden Methoden genutzt, so dass ein korrekter Zugriff auf die Dialogvorlagen ermöglicht wird.

Hinzufügen	
Von	*HINSTANCE hInstance;* und entsprechende Zugriffsfunktion *HINSTANCE GetInstance()*
Dateien	include\swintrac\swintrac.h, swintrac\dlg_tpl.cpp
Klassen	TModalDlg und TModelessDlg

Hinzufügen	
von	*TModalDlg (const char*, HINSTANCE hInstance);* *TModelessDlg (const char*, TModelessDlg** , HINSTANCE hInstance);* - entsprechende zusätzliche Konstruktoren mit Instanzhandle als Parameter
Dateien	include\swintrac\swintrac.h, swintrac\dlg_tpl.cpp
Klassen	TModalDlg und TModelessDlg

Ersetzen	
von	int result= DialogBoxParam(hModuleInstance, GetResName(),…
durch	int result= DialogBoxParam(hInstance, GetResName(),...
Methode	TModalDlg::ExecuteDialog()
Dateien	include\swintrac\swintrac.h, swintrac\dlg_tpl.cpp

Ersetzen	
von	*if (aInstance == 0) aInstance = hModuleInstance;*
durch	*if (aInstance == 0) aInstance = GetMainInstance();*
Methode	*TModelessDlg::Initialize ()*
Dateien	include\swintrac\swintrac.h, swintrac\dlg_tpl.cpp

Ersetzen	
von	alle Konstruktoraufrufe von *TModalDlg* und *TModelessDlg*
durch	die neuen Konstruktoren ersetzen (in allen Projektdateien)
Dateien	include\swintrac\swintrac.h, swintrac\dlg_tpl.cpp

Auch hier wurden die entsprechenden Quelltextfragmente durch eine semantische Teilanalyse des Portierungsgegenstandes lokalisiert.

Direkte Hardwarezugriffe

Um XCTL sinnvoll nutzen zu können, ist der Zugriff auf anwendungsspezifische Hardwarekomponenten notwendig. Aufgrund der Windows-NT-eigenen Sicherheitsmechanismen ist ein direkter Hardwarezugriff für Anwendungsprogramme unter Windows 2000 allerdings nicht durchführbar. Faktisch sind Hardwarezugriffe nur über die Systemschnittstellen möglich. Hierfür ist aber i. A. die Anwesenheit hardwarespezifischer Softwaremodule (Gerätetreiber) erforderlich, welche die Hardwarezugriffe implementieren. Stehen solche Treiber nicht schon zur Verfügung, müssen diese selbst erstellt werden.

Die hier für die Anpassung der Hardwareansteuerung angedeuteten notwendigen Transformationen werden im folgenden Abschnitt präzisiert und erläutert.

Adaption der Hardwareansteuerung

Da unter Windows 2000 keine direkten Zugriffe auf die Hardware aus einem Anwendungsprogramm heraus möglich sind, muss für XCTL ein anderer Weg gefunden werden, mit der anwendungsspezifischen Hardware zu kommunizieren. Microsoft schreibt in einem solche Falle die Verwendung eines Gerätetreibers zur Hardwareansteuerung vor. Leider wird die verwendete Hardware von Windows 2000 jedoch nicht standardmäßig unterstützt und die Hersteller[2] der von XCTL verwendeten Spezialhardware bieten keine entsprechenden Treiber an. Damit ergeben sich für die Lösung dieses Problems folgende Möglichkeiten:

(1) Erstellung eines eigenen Windows-2000-konformen Gerätetreibers der im Kernelmodus arbeitet.

(2) Verwendung eines generischen Gerätetreibers, der Zugriff auf die gewünschten I/O-Ports gewährt.

(3) Verwendung eines Kernelmodustreibers, der den Zugriffsschutz auf I/O-Ports für den gesamten I/O-Adressbreich ausschaltet.

Da die letzten beiden Varianten keine Möglichkeit bieten, den vom Motorcontroller C-812 verwendeten Hardwarezugriffsmechanismus *Memory-mapped-I/O* für XCTL zur Verfügung zu stellen, scheiden diese von vornherein aus. Aber auch die Tatsache, dass mit diesen Varianten die Sicherheitsmechanismen von Windows 2000, die eine wichtige Voraussetzung für die Stabilität dieses Betriebssystems darstellen, umgangen werden, muss zu der Entscheidung führen, derartige „Workarounds“ nicht zu verwenden. Aus diesen Gründen werden Windows 2000 konforme Gerätetreiber entwickelt, die im Kernelmodus arbeiten und die Kommunikationsschnittstelle zwischen XCTL und Messplatzhardware bilden.

[2] [Axiom 92], [Braun 94], [PI 95], [Radicon 95]

Die Adaption der Hardwareansteuerung besteht aus den folgenden Schritten:

1. Identifikation aller Quelltextstellen, an denen Zugriffe auf die Hardware durchgeführt werden. Da die bisherige Analyse ergeben hat, dass die zwei Mechnismen *Port-I/O* und *Memory-Mapped-I/O* Verwendung finden, muss nach Quelltextfragmenten gesucht werden, die entsprechende Zugriffe implementieren. Zugriffe auf *Ports* mittels *in* und *out*, bzw. deren Hochsprachenäquivalenten *inp*() und *outp()* sind im Quelltext relativ leicht mittels Textsuche zu lokalisieren. Zugriffe auf *mapped-Memory* sind nur sehr schwer zu identifizieren, da die Verwendung von Zeigern in einer Sprache wie C++ allgemein üblich ist. Um derartige Programmstellen zu identifizieren, bedarf es einer semantischen Quelltextanalyse. Dabei können aussagekräftige Zeigerbezeichnungen eine große Unterstützung sein.

2. Spezifikation und Erstellung der benötigten Gerätetreiber.

3. Sind alle relevanten Programmstellen lokalisiert, müssen diese direkten Hardwarezugriffe durch indirekte ersetzt werden. Indirekt bedeutet in diesem Falle, dass ein entsprechender IOCTL-Code an einen bestimmten Treiber mittels der API-Funktion *DeviceIoControl()* gesendet wird. Da es hierzu noch verschiedener Verwaltungsaufrufe bedarf, muss zusätzlich die interne Struktur der betroffenen Objekte geändert werden.

Da die Umstellung der Harwareansteuerung also umfangreiche Quelltextanpassungen und entsprechend detaillierte Kenntnisse des Portierungsgegenstandes erfordert, ist an dieser Stelle eine genaue Analyse von XCTL unumgänglich.

Analyse von XCTL - Allgemein

Die Quellen des Portierungsgegenstandes umfassen derzeit nach umfangreichen Änderungen durch Mitglieder der Projektgruppe[3] ca.

[3] siehe: https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98y/management/personen.html

50.000 LOC in ca. 140 Dateien. Das Programm wird als 16-Bit Applikation bis jetzt weiterentwickelt. Diese Version von XCTL greift zur Steuerung der Hardware auf verschiedene Mechanismen zurück, welche unter Verwendung von direkten I/O-Zugriffen mittels der Bibliotheksfunktionen *inp()/outp()*, Zeigerarithmetik und inline-Assembler-Anweisungen implementiert werden. Neben diesen in der Win16-Programmierung üblichen Vorgehensweisen werden zur Hardwareansteuerung auch DLLs genutzt, die von den Herstellern der Hardware zur Verfügung gestellt werden. Diese 16-bit-DLLs bieten hardwarebezogene Spezialfunktionalität, die im weitesten Sinne mit der eines Gerätetreibers verglichen werden könnte. Tabelle [5.1] zeigt die verwendeten DLLs im Einzelnen. Diese Module wurden nicht von den Autoren von XCTL erstellt, so dass entsprechend auch keine Quelltexte vorliegen.

Von XCTL verwendete, fremderstellte DLLs		
DLL	**Hersteller**	**Bedeutung**
ASA.DLL	Hecus M.Braun, Östereich	dient der Kommunikation mit der ASA-Steuerkarte des Braun-PSD verwendete Funktionen: -GetPort() -SetPort() -SetTimeOut() -GetData()
Win488.dll	unbekannt	liefert Funktionen für die IEEE488 Schnittstelle

Tabelle [5.1]

Da diese 16-bit-DLLs nicht unter Win32 ausgeführt werden können, muss die entsprechende Funktionalität anderweitig zur Verfügung gestellt werden.

Weiterführende Informationen werden durch eine Analyse der Architektur des Portierungsgegenstandes gewonnen.

Analyse von XCTL – Architektur

Die hier zusammengestellten Informationen sind Ergebnis einer semantischen Analyse der Portierungsbasis.

Das XCTL–System besteht aus 14 Subsystemen, die sich auf sieben Softwaremodule verteilen, wobei die Funktionalität einiger Subsysteme auf verschiedene Module verteilt implementiert wurde. Grundlage der weiteren Erläuterung der Systemarchitektur ist die von Kay Schützler in seiner Diplomarbeit [Schützler01] erarbeitete Subsystemstruktur. Diese Struktur wurde bis zum Zeitpunkt der Portierung durch zusätzlich implementierte Funktionalität und Architekturänderungen erweitert und verändert.
Die semantische Analyse von XCTL im Bereich der Detektoren wurde bereits in [HaPaPi 00] durchgeführt und konnte an dieser Stelle genutzt werden.

Tabelle [5.2] zeigt die Subsysteme und ihren Zuordnung zu den Softwaremodulen.

XCTL-Subsysteme		
	Subsystem	**Modul**
1	Motorsteuerung	Motors.dll
2	Detektornutzung	Counters.dll
3	Repräsentation und Darstellung der Messdaten	Splib.dll, XControl.exe
4	Topographie	XControl.exe
5	Diffraktometrie/Reflektometrie	XControl.exe
6	Ablaufsteuerung	XControl.exe
7	Online-Hilfe	Sphelp.hlp
8	Interaktion mit der Software	Splib.dll, XControl.exe
8	Interne Funktionalität und allgemeine Definitionen	Splib.dll, XControl.exe
10	Windows-Ressourcen	Motors.dll, Counters.dll, Splib.dll, Protocol.dll, XControl.exe
11	Automatische Justage	XControl.exe
12	Protokollbuch	Protocol.dll
13	Hardwareansteuerung	Hwio.dll
14	Allgemeine Einstellungen	XControl.exe

Tabelle [5.2]

Für ein besseres Verständnis des internen Aufbaus des XCTL-Systems werden im Folgenden die Subsysteme in ihrer Funktion und Abbildung auf Quelldateien kurz erläutert.

(1) Motorsteuerung
Das Subsystem stellt eine logische Abstraktionsschicht zur Ansteuerung und Kontrolle der Motoren des Messplatzes zur Verfügung. Zu diesem

Zweck werden Klassen implementiert, deren Instanzen die Hardwareobjekte (physisch vorhandene Motoren) repräsentieren.

Modul	Motors.dll
Header	C8x2.h, ieee.h, m_layer.h, m_motcom.h, m_mothw.h, motrstrg.h, msimstat.h
Implementation	m_layer.cpp, motors.cpp, msimstat.cpp

(2) Detektornutzung
Das Subsystem stellt eine logische Abstraktionsschicht zur Ansteuerung und Kontrolle der Detektoren des Messplatzes zur Verfügung. Zu diesem Zweck werden Klassen implementiert, deren Instanzen die Hardwareobjekte (physisch vorhandene Detektoren) repräsentieren .

Modul	Counters.dll
Header	detec_hw.h, detecgui.h, detecmes.h, detecuse.h, range.h
Implementation	detec_hw.cpp, detecgui.cpp, detecmes.cpp, detecuse.cpp, dllmain.cpp

(3) Repräsentation und Darstellung der Messdaten
Das Subsystem stellt Klassen für die Datenhaltung (Datenstrukturen) und Datendarstellung (Fenstervorlagen) zur Verfügung.

Modul	Splib.dll, XControl.exe
Header	datavisa.h, m_data.h, swintrac.h
Implementation	m_curve.cpp, m_data.cpp

(4) Topographie
Das Subsystem stellt Klassen für die Funktionalität des Messvorganges *Topographie* und die zugehörige Nutzerinteraktion zur Verfügung.

Modul	XControl.exe
Header	topogrfy.h, tp_funk.h, tp_gui.h
Implementation	m_topo.cpp, tp_funk.cpp, tp_gui.cpp

(5) Diffraktometrie/Reflektometrie
Das Subsystem stellt Klassen für die Funktionalität der Messvorgänge *Diffraktometrie* und *Reflektometrie* und die zugehörige Nutzerinteraktion zur Verfügung.

Modul	XControl.exe
Header	difrkmty.h, m_arscan.h, m_ccdscn.h, m_dlgdif.h, m_scan.h
Implementation	m_arscan.cpp, m_ccdscn.cpp, m_dlgdif.cpp, m_scan.cpp

(6) Ablaufsteuerung
Das Subsystem stellt Klassen für die Steuerung eines Programms über eine Skriptsprache und Funktionen zur Messungsvorbereitung (z.B. manuelle Justage) zur Verfügung.

Modul	XControl.exe
Header	m_steerg.h, workflow.h, mj_funk.h, mj_gui.h, mj_ofunk.h, mj_ogui.h, mj_old.h
Implementation	m_steerg.cpp, mj_funk.cpp, mj_gui.cpp, mj_ofunk.cpp, mj_ogui.cpp, mj_old.cpp

(7) Online-Hilfe
Das Subsystem stellt Dokumente für die Online-Hilfe zur Verfügung. Aus den Quelldateien wird mit Hilfe des *Microsoft Help Compiler* bzw. *Help Workshop* eine Windows-Hilfedatei erzeugt.

Modul	Sphelp.hlp
Header	help_def.h
Implementation	sphelp.hsc, sphelp16/32.hpj, sphelp.rtf

(8) Interaktion mit der Software (Benutzeroberfläche)
Das Subsystem stellt Klassen zur Nutzerinteraktion (Fenster- und Dialoge) zur Verfügung.

Modul	Splib.dll, XControl.exe
Header	dgl_tpl.h, m_dlg.h, swintrac.h
Implementation	dgl_tpl.cpp, m_device.cpp, m_dlg.cpp

(9) Interne Funktionalität & allgemeine Definitionen
Das Subsystem stellt Kontrollstrukturen für den allgemeinen Programmablauf zur Verfügung (Programmrumpf).

Modul	Splib.dll, XControl.exe
Header	evrythng.h, u_files.h, u_timer.h, u_values.h
Implementation	l_layer.cpp, m_main.cpp, u_files.cpp, u_timer.cpp, u_values.cpp

(10) Automatische Justage
Das Subsystem stellt Klassen und Funktionen zur automatischen Justierung einer Messprobe zur Verfügung.

Modul	XControl.exe
Header	Autojust.h, matrix.h, m_justag.h, transfrm.h
Implementation	matrix.cpp, m_justag.cpp, transfrm.cpp

(11) Windows-Ressourcen
Das Subsystem stellt Ressourcen für das grafische Nutzerinterface zur Verfügung.

Modul	Motors.dll, Counters.dll, Splib.dll, XControl.exe, Protocol.dll
Header	rc_def.h, rc_prtow.h
Implementation	counters.rc, main.rc, motors.rc, splib.rc, protocol.rc, scanner.rc, splib.rc

(12) Protokollbuch
Dieses Subsystem stellt ein elektronisches Protokollbuch zur Verfügung.

Modul	Protocol.dll
Header	p_lang.cpp, prdipara.h, prdiprnw.h, procombw.cpp, proprnw.h, protdif.h, protdifw.h, proto.h, protocol.cpp, protow.h, protpasw.h, prottop.h, prottopw.h, prtopara.h, prtoprnw.h
Implementation	mespara.cpp, p_lang.cpp, prdiprnw.cpp, procombw.cpp, proprnw.cpp, protdif.cpp, protdifw.cpp, proto.cpp, protocol.cpp, protow.cpp, protpasw.cpp, prottop.cpp, prottopw.cpp, prtoprnw.cpp

(13) Hardwareansteuerung
Das Subsystem stellt allgemeine Klassen und Funktionen für den Zugriff auf Hardwarekomponenten zur Verfügung (Port-I/O).

Modul	Hwio.dll
Header	hwio.h
Implementation	hwio.cpp

(14) Allgemeine Einstellungen
Dieses Subsystem stellt Klassen und Funktionen zur Erfassung und Verwaltung allgemeiner Einstellungen des Programms zur Verfügung.

Modul	XControl.exe
Header	mespara.h, mesparaw.h
Implementation	mespara.cpp, mesparaw.h

Aus den Informationen der semantischen Analyse wurde die in Abb. [5.1] dargestellte Architektur des Portierungsgegenstandes ermittelt. Deutlich ist die Verletzung der Schichtentrennung bei Hardwarezugriffen zu erkennen. Dieser Umstand lässt einen relativ hohen Portierungsaufwand bezüglich der Adaption der Hardwareansteuerung erwarten, da die Identifizierung anzupassender Quelltextfragmente erschwert wird.

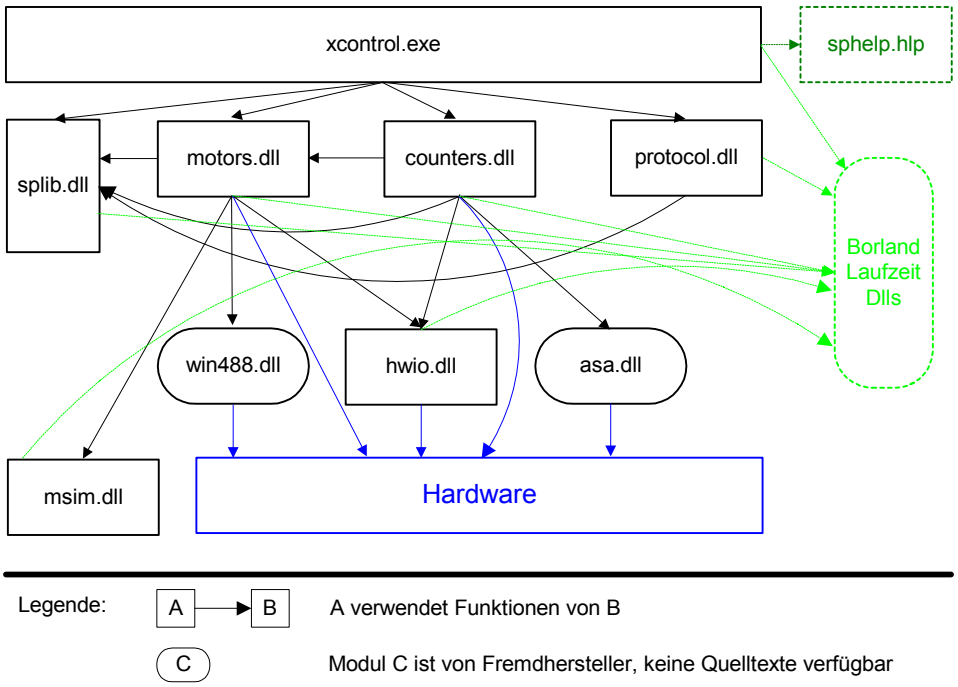


Abbildung [5.1] Architektur von XCTL (Portierungsbasis)

Für die Adaption der Hardwareansteuerung scheinen ausschließlich die XCTL-Module *Motors.dll*, *Counters.dll* und *Hwio.dll* von Relevanz zu sein. Die Implementierungsdetails dieser Module sind im weiteren Verlauf genauer zu untersuchen. Zusätzlich muss die in *Win488.dll* und *Asa.dll* implementierte Funktionalität analysiert werden.

Analyse – Motors.dll (Motorsteuerung)

Die Analyse des Subsystems *Motorsteuerung* zeigt, dass die wesentliche Funktionalität der Hardwareansteuerng bezüglich der Motoren in den folgenden Klassen und modulglobalen Funktionen implementiert ist. Entsprechend sind auch dort die notwendigen Portierungsanpassungen vorzunehmen.

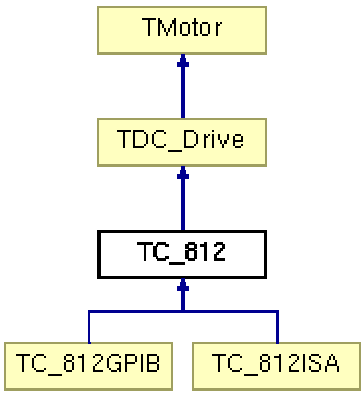
Klassen und Funktionen		
Klassenname/ Funktionsname	**Parent**	**Funktion**
TC_812	TDC_Drive	Zugriff auf einen an die Motorsteuerungskarte C-812 angeschlossenen Motor (Basisklasse) Semantischer Fehler[4]: eigentlich sollte die Klasse den Zugriff auf die Steuerungskarte kapseln und nicht auf einen angeschlossenen Motor
TC_812GPIB	TC_812	Zugriff auf einen an die Motorsteuerungskarte C-812 angeschlossenen Motor (Ansteuerung über IEEE488-Schnittstelle)
TC_812ISA	TC_812	Zugriff auf einen an die Motorsteuerungskarte C-812 angeschlossenen Motor (Ansteuerung über ISA-Schnittstelle)
TC_832ISA	TC_812	Zugriff auf einen an die Motorsteuerungskarte C-832 angeschlossenen Motor (Ansteuerung über ISA-Schnittstelle) Semantischer Fehler[4]: eigentlich sollte die Klasse den Zugriff auf die Steuerungskarte kapseln und nicht auf einen angeschlossenen Motor
C812ISA_Get()		Datentransfer C-812 (Lesen)
C812ISA_Put()		Datentransfer C-812 (Schreiben)
C832_Get()		Datentransfer C-832 (Lesen)
C832_Put()		Datentransfer C-832 (Schreiben)
GetDWord()		Datentransfer C-832 (Lesen)
GetWord()		Datentransfer C-832 (Lesen)
PutDWord()		Datentransfer C-832 (Schreiben)
PutWord()		Datentransfer C-832 (Schreiben)
Drive628c()		Kommandofunktion für C-832
LM628Ready()		Warten auf Bereitschaft der Steuerkarte C-832

Tabelle [5.3]

Für ein besseres Verständnis der internen Abläufe werden die erwähnten Klassen im Folgenden genauer analysiert und beschrieben.
Anmerkung:
Die Tabellen stellen unter anderem die Aufrufbeziehungen der Methoden der einzelnen Klassen dar. Ein fehlender Eintrag in der Zeile „wird verwendet von“ bedeutet nicht, dass die betreffende Methode nie aufgerufen wird. In diesem Fall wird die Methode in der Implementation einer Basisklasse aufgerufen. Wenn eine Methode tatsächlich nie aufgerufen wird und somit toten Code darstellt, wird in jedem Falle explizit darauf hingewiesen.

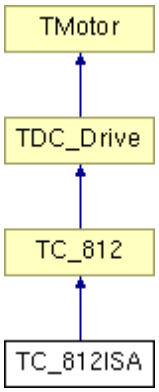
[4] siehe Kap. Fehlerbeschreibung

(1) Klasse TC_812



Diese Klasse hat die Aufgabe, die Funktionalität der an eine Motorcontrollerkarte vom Typ *C-812* angeschlossenen Motoren gegenüber XCTL zu kapseln. Da diese Karte über zwei verschiedene Hardwareschnittstellen angesteuert werden kann, werden die Hardwarezugriffe nicht in dieser Klasse implementiert, sondern in den entsprechend abgeleiteten Klassen *TC_812GPIB* bzw. *TC_812ISA*. Aus diesem Grund werden in dieser Klasse nur geringfügige Portierungsanpassungen notwendig. Die Klasse *TC_812GPIB*, welche Zugriffe über den Schnittstellentyp *IEEE 488* kapselt, wird im Zielsystem nicht mehr benötigt und deshalb hier nicht weiter betrachtet.

(2) Klasse TC_812ISA



Diese Klasse kapselt die eigentlichen Hardwarezugriffe über das ISA-Hardwareinterface der Motorsteuerkarte *C-812*. Die Steuerkarte wird mittels Zeigerarithmetik angesteuert, da die Hardwarekommunikation über *Memor- mapped- I/O* implementiert ist.

private Attribute		
Typ	Attribut	Beschreibung
WORD	**wBaseAddr**	Basisadresse aus hardware.ini, dient der Berechnung der folgenden Adressen
LPSTR	**lpFlag**	Adresse für Statusregister
LPSTR	**lpIn**	Adresse für Inputregister
LPSTR	**lpOut1**	Adresse für Outputregister 1
LPSTR	**lpOut2**	Adresse für Outputregister 2
LPSTR	**lpDPRam**	Basisadresse für Memory Mapped IO

öffentliche Methoden	
TC_812ISA (void)	
verwendet wird	::GetHWFile(),TMotor::nId, TC_812::nWaitTicks lpDPRam, lpFlag, lpIn, lpOut1, lpOut2, wBaseAddr,:: logstream
Beschreibung	Konstruktor

geschützte Methoden	
int **ExecuteCmd** (char *)	
verwendet wird	::Delay(), GetChar(), PutChar() TMotor::bControlBoardOk,
wird verwendet von	CheckBoardOk()
Beschreibung	sendet den übergebenen Kommandostring an die Controllerkarte

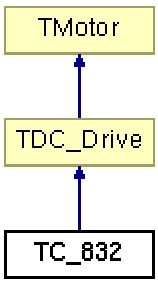
BOOL **CheckBoardOk** (void)	
verwendet wird	C812ISA_Get(), C812ISA_Put(), ExecuteCmd() lpIn, lpOut1, wBaseAddr
wird verwendet von	
Beschreibung	testet, ob der Motorcontroller vorhanden und funktionsbereit ist
BOOL **SetHome** (void)	
verwendet wird	::C812ISA_Put(), TC_812::SetHome() lpDPRam, TC_812::nOnBoardId
wird verwendet von	
Beschreibung	legt Homeposition fest
BOOL **IsMoveFinish** (void)	
verwendet wird	_GetFailure(), ::DelayTime(), TC_812::IsMoveFinish(), TDC_Drive::wDeathBand
wird verwendet von	
Beschreibung	ermittelt, ob der Motor seine Bewegung beendet hat
void **StartCheckScan** (void)	
verwendet wird	::mSavePosition(),TDC_Drive::MoveByPosition(), TMotor::wPositionWidth ::nCallBackAction, ::nEvent, ::nTimeTicks
wird verwendet von	

Beschreibung	Bedeutung unklar
BOOL **_GetPosition** (long *)	
verwendet wird	TC_812::_GetPosition(), ::C812ISA_Get() LpDPRam, TC_812::nOnBoardId
wird verwendet von	
Beschreibung	ermittelt die aktuelle Motorposition
BOOL **_GetFailure** (long *)	
verwendet wird	::C812ISA_Get(), TC_812::_GetFailure() lpDPRam, TC_812::nOnBoardId
wird verwendet von	IsMoveFinish()
Beschreibung	ist BOOL **_GetPosition** (long *) ähnlich, ermittelt die Positionsabweichung

private Methoden	
int **PutChar** (const char)	
verwendet wird	::C812ISA_Get(), ::C812ISA_Put() lpFlag, lpOut1, lpOut2
wird verwendet von	ExecuteCmd()
Beschreibung	sendet ein Zeichen an die Controllerkarte mit der globalen Funktion *C812ISA_Put()*, ermittelt Status der Controllerkarte mit der globalen Funktion *C812ISA_Get()*

char **GetChar** (void)	
verwendet wird	::C812ISA_Get(), :: Delay() lpFlag, lpIn
wird verwendet von	ExecuteCmd()
Beschreibung	liest ein Zeichen von der Controllerkarte mit der globalen Funktion *C812ISA_Get()*

(3) Klasse TC_832



Aufgabe dieser Klasse ist die Kapselung der Hardwarezugriffe bezüglich der Motorsteuerkarte *C-832*. Da bei dieser Karte *Port-I/O* implementiert wurde, erfolgen die Hardwarezugriffe über die Funktionen *inp()* bzw. *outp()*, teilweise auch indirekt über die Klasse *TIOPort*.

öffentliche Attribute		
Typ	Attribut	Beschreibung
void CALLBACK (*lpfnLimitWatch) (CALLBACKUINT, UINT, DWORD, DWORD, DWORD)	**lpfnLimitWatch**	Zeiger auf Funktion LimitWatch

geschützte Attribute		
Typ	Attribut	Beschreibung
WORD	**wKI**	IntegralGain
WORD	**wKL**	IntegralLimit
WORD	**wKP**	StaticGain
WORD	**wKD**	DynamicGain

private Attribute		
Typ	Attribut	Beschreibung
WORD	**wBaseAddr**	Basisadresse aus hardware.ini, dient der Berechnung der folgenden Adressen
Int	**nOnBoardId**	identifiziert den Motorcontroller auf der Steuerkarte
BYTE	**cConfig**	speichert Konfigurationsflags der Motorcontroller
WORD	**wSample**	Bedeutung unklar

statische private Attribute		
Typ	Attribut	Beschreibung
BOOL	**bLimitHit**	Statusflag, welches anzeigt, ob Limit erreicht ist
BOOL	**bIdle**	Statusflag, welches anzeigt, dass Motor in Benutzung ist
BOOL	**bIOActive**	unbenutzt

öffentliche Methoden	
TC_832 (void)	
verwendet wird	::GetHWFile() ,cConfig, TMotor::nId, nOnBoardId, wBaseAddr, wKD, wKI, wKL, wKP, wSample
Beschreibung	Konstruktor
BOOL **StopLimitWatch** (void)	
verwendet wird	TMotor::bLimitWatchActive, :: nEvent
wird verwendet von	
Beschreibung	löscht einen Timer

BOOL **StartLimitWatch** (void)	
verwendet wird	TMotor::bLimitWatchActive, LimitWatch() :: nEvent
wird verwendet von	
Beschreibung	setzt einen Timer
void **OptimizingDlg** (void)	
verwendet wird	TModalDlg::ExecuteDialog(), ::GetFrameHandle(), TOptimizeDC_832Dlg
wird verwendet von	
Beschreibung	erzeugt einen Dialog vom Typ *TOptimizeDC_832Dlg*

geschützte Methoden	
int **SaveSettings** (BOOL)	
verwendet wird	
wird verwendet von	TOptimizeDC_832Dlg::CanClose()
Beschreibung	sichert bestimmte Einstellungen in der Datei *hardware.ini*
BOOL **ActivateFilterParameters** (void)	
verwendet wird	Drive628(), SetAcceleration(), SetVelocity(),TMotor::dwAcceleration, TMotor::dwMaxVelocity, TDC_Drive::GetSpeed() wKD, wKI, wKL, wKP, wSample
wird verwendet von	TOptimizeDC_832Dlg::Dlg_OnCommand(), TOptimizeDC_832Dlg::LeaveDialog()
Beschreibung	sendet bestimmte Befehle an einen der beiden Controller der Motorsteuerungskarte C-823 mit der globalen Funktion *Drive628c()*, damit werden Parameter für die weitere Arbeit mit dem Controller gesetzt
BOOL **CheckBoardOk** (void)	
verwendet wird	::C832_Get(), ::C832_Put(), TMotor::bControlBoardOk cConfig, nOnBoardId, wBaseAddr
wird verwendet von	
Beschreibung	testet ob der Motorcontroller vorhanden und funktionsbereit ist
BOOL **ActivateDrive** (void)	
verwendet wird	_GetPosition(), Drive628()
wird verwendet von	
Beschreibung	aktiviert den Motor, mit der globalen Funktion *Drive628c()* werden Parameter/Befehle an den entsprechenden Controller übermittelt
void **StartCheckScan** (void)	
verwendet wird	TDC_Drive::MoveByPosition(), ::mSavePosition() ::nCallBackAction, ::nEvent, ::nTimeTicks, TMotor::wPositionWidth
wird verwendet von	
Beschreibung	ein Timer wird gestartet, die Motorposition wird verändert, Bedeutung unklar

int **Reset** (void)	
verwendet wird	Drive628()
wird verwendet von	
Beschreibung	setzt Position auf Null, führt Reset des entsprechenden Controllers durch

private Methoden	
long **Drive628** (BYTE, WORD, long)	
verwendet wird	::Drive628c() ::baddr, cConfig nOnBoardId, raddr, wBaseAddr
wird verwendet von	_GetFailure(), _GetPosition(), ActivateDrive(), ActivateFilterParameters(), GetStatus(), IsIndexArrived(), IsLimitHit(), IsMoveFinish(), MoveAbsolute(), MoveRelative(), Reset(), SetAcceleration(), SetHome(), SetVelocity(), StartToIndex(), StopDrive()
Beschreibung	initialisiert globale Variablen, ruft globale Funktion *::Drive628c()* auf
int **ExecuteCmd** (LPSTR)	
verwendet wird	-
wird verwendet von	-
Beschreibung	keine Bedeutung in dieser Klasse, wird dementsprechend nie verwendet
BOOL **IsMoveFinish** (void)	
verwendet wird	::DelayTime(), Drive628()
wird verwendet von	IsIndexArrived(), IsLimitHit(), StopDrive()
Beschreibung	testet, ob Bewegung beendet
BOOL **IsLimitHit** (void)	
verwendet wird	::C832_Get(), ::C832_Put(), ::DelayTime(), Drive628(), IsMoveFinish(), ::LM628Ready(), nOnBoardId, wBaseAddr, bLimitHit, TMotor::bRangeHit, TMotor::dwRemoveLimit, cConfig, TMotor::bUpwards
wird verwendet von	IsIndexArrived()
Beschreibung	testet, ob Motor Limit erreicht hat
BOOL **IsIndexArrived** (void)	
verwendet wird	::Delay(), Drive628(), IsLimitHit(), IsMoveFinish(), TMotor::bIndexDetected, TMotor::bIndexLine, TMotor::bMoveFirstToLimit, TMotor::bRangeHit, TMotor::bUpwards
wird verwendet von	
Beschreibung	testet, ob Motor Index erreicht hat
BOOL **StartToIndex** (long &)	
verwendet wird	::DelayTime(), Drive628(), TMotor::SetCalibrationState(), TMotor::SetCorrectionState(), TDC_Drive::SetSpeed(), TMotor::bIndexDetected, TMotor::bIndexLine, bLimitHit, TMotor::bMoveFirstToLimit, TMotor::bRangeHit, TMotor::bUpwards , TMotor::lDeltaPosition, TMotor::lPosition

wird verwendet von	
Beschreibung	Bedeutung unklar
BOOL **StopDrive** (BOOL)	
verwendet wird	::DelayTime(), Drive628(), IsMoveFinish()
wird verwendet von	
Beschreibung	stoppt den Motor
BOOL **SetLimit** (DWORD)	
verwendet wird	TMotor::dwRemoveLimit
wird verwendet von	
Beschreibung	Accessorfunktion für Basisklassenmember *TMotor::dwRemoveLimit*
BOOL **SetVelocity** (DWORD)	
verwendet wird	Drive628(), TMotor::dwMaxVelocity, TMotor::dwVelocity
wird verwendet von	ActivateFilterParameters()
Beschreibung	Beschleunigung festlegen
DWORD **GetVelocity** (void)	
verwendet wird	TMotor::dwMaxVelocity und TMotor::dwVelocity
wird verwendet von	
Beschreibung	liefert einen Geschwindigkeitswert
BOOL **SetHome** (void)	
verwendet wird	Drive628(), TMotor::SetHome()
wird verwendet von	
Beschreibung	legt Homeposition fest
BOOL **SetAcceleration** (DWORD)	
verwendet wird	Drive628() TMotor::dwAcceleration
wird verwendet von	ActivateFilterParameters()
Beschreibung	legt Beschleunigung fest
DWORD **GetAcceleration** (void)	
verwendet wird	TMotor::dwAcceleration
wird verwendet von	
Beschreibung	Accessorfunktion für Basisklassenmember TMotor::dwAcceleration
WORD **GetStatus** (void)	
verwendet wird	Drive628()
wird verwendet von	
Beschreibung	ermittelt den Status des Controllers
BOOL **_GetPosition** (long *)	
verwendet wird	Drive628()
wird verwendet von	ActivateDrive()
Beschreibung	ermittelt die Postion des Motors
BOOL **_GetFailure** (long *)	
verwendet wird	Drive628()
wird verwendet von	
Beschreibung	Bedeutung unklar, vermutlich wird Positionsabweichung

	ermittelt
BOOL **MoveRelative** (long)	
verwendet wird	Drive628()
wird verwendet von	
Beschreibung	bewegt den Motor relativ zur aktuellen Postion an neue Position
BOOL **MoveAbsolute** (long)	
verwendet wird	Drive628()
wird verwendet von	
Beschreibung	bewegt den Motor zur neuen Postion

statische private Methoden	
void CALLBACK **LimitWatch** (UINT, UINT, DWORD, DWORD, DWORD)	
verwendet wird	::baddr, bLimitHit, :: C832_Get(), ::C832_Put, ::config, ::Drive628c(), ::GetFrameHandle(), ::LM628Ready(), ::raddr
wird verwendet von	StartLimitWatch()
Beschreibung	Callback-Funktion für Timer

(4) Globale Funktionen

Die nachfolgend aufgeführten modulglobalen Funktionen implementieren teilweise Hardwarezugriffe der oben genannten Klassen.

inline static char **C812ISA_Get**(char* addr)	
verwendet wird	c812isa_get_callback()
wird verwendet von	TC_812ISA::_GetFailure(), TC_812ISA::_GetPosition(), TC_812ISA::CheckBoardOk(), TC_812ISA::GetChar(), TC_812ISA::PutChar()
Beschreibung	liest ein Byte vom Motorcontroller C-812 über direkten Speicherzugriff

inline static void **C812ISA_Put**(char* addr, char c)	
verwendet wird	c812isa_put_callback()
wird verwendet von	TC_812ISA::CheckBoardOk(), TC_812ISA::PutChar(), TC_812ISA::SetHome()
Beschreibung	sendet ein Byte an den Motorcontroller C-812 über direkten Speicherzugriff

inline static int **C832_Get**(unsigned port)	
verwendet wird	c832_get_callback(), TIOPort::In8Bit()
wird verwendet von	TC_832::CheckBoardOk(), Drive628c(), GetDWord(), GetWord(), TC_832::IsLimitHit(), TC_832::LimitWatch(), LM628Ready()
Beschreibung	liest ein Byte vom Motorcontroller C-832 über *TIOPort::In8Bit()*

inline static void **C832_Put**(unsigned port, int value)	
verwendet wird	c832_put_callback(), TIOPort::Out8Bit()
wird verwendet von	TC_832::CheckBoardOk(), Drive628c(), GetDWord(), GetWord(), TC_832::IsLimitHit(), TC_832::LimitWatch(), LM628Ready(), PutDWord(), PutWord()
Beschreibung	sendet ein Byte an den Motorcontroller C-832 über *TIOPort::Out8Bit()*

long **GetDWord**(WORD, WORD)	
verwendet wird	C832_Get(), C832_Put(), LM628Ready()
wird verwendet von	Drive628c()
Beschreibung	liest DWORD vom Motorcontroller C-832

int **GetWord**(WORD, WORD)	
verwendet wird	C832_Get(), C832_Put(), LM628Ready()
wird verwendet von	Drive628c()
Beschreibung	liest WORD vom Motorcontroller C-832

void **PutDWord**(long, WORD, WORD)	
verwendet wird	C832_Put(), LM628Ready().
wird verwendet von	Drive628c()
Beschreibung	sendet DWORD an den Motorcontroller C-832

void **PutWord**(int, WORD, WORD)	
verwendet wird	C832_Put(), Delay(), LM628Ready()
wird verwendet von	Drive628c()
Beschreibung	sendet WORD an den Motorcontroller C-832

long **Drive628c** (BYTE cmd,WORD ctrl_word,long param,WORD base, WORD regaddr)	
verwendet wird	TC_832::bIdle, C832_Get(), C832_Put(), TCSet::cmd, TCSet::data, GetDWord(), GetWord(), TCSet::length32, LM628Ready(), PutDWord(), PutWord(), TCSet::report
wird verwendet von	TC_832::LimitWatch(),TC_832::Drive628()
Beschreibung	Kommandofunktion für den Controller auf der Motorsteuerkarte C-832

BOOL **LM628Ready**(WORD)	
verwendet wird	C832_Get(), C832_Put()
wird verwendet von	Drive628c(), GetDWord(), GetWord(), TC_832::IsLimitHit(), TC_832::LimitWatch(), PutDWord(), PutWord()
Beschreibung	wartet darauf, dass der Controller auf der Motorsteuerkarte C-832 Bereitschaft signalisiert

Analyse – Counters.dll (Detektornutzung)

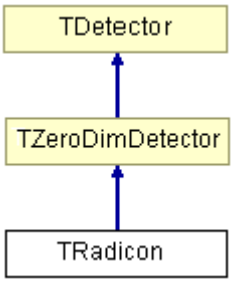
Die Analyse des Subsystems *Detektornutzung* zeigt, dass die wesentliche Funktionalität der Hardwareansteuerng bezüglich der Detektoren in den folgenden Klassen implementiert ist. Entsprechend sind auch dort die notwendigen Portierungsanpassungen vorzunehmen.
Für ein besseres Verständnis der internen Abläufe werden die erwähnten Klassen im Folgenden genauer analysiert und beschrieben.

Klassen		
Klassenname	**Parent**	**Funktion**
TRadicon	TZeroDimDetector	Funktionalität des Detektors Radicon SCSCS
TRadiconHW	-	Hardwarezugriffe für Radicon SCSCS
TGenericDetector	TZeroDimDetector	Funktionalität des Detektors Generic SCSCS
TAm9513	-	Hardwarezugriffe für Detektor Generic SCSCS
TBraunPSD	TOneDimDetector	Funktionalität und Hardwarezugriffe des Detektors Braun-PSD
TStoePSD	TOneDimDetector	Funktionalität und Hardwarezugriffe des Detektors Braun-PSD

Tabelle [5.4]

Anmerkung:
Die Tabellen stellen unter anderem die Aufrufbeziehungen der Methoden der einzelnen Klassen dar. Ein fehlender Eintrag in der Zeile „wird verwendet von“ bedeutet nicht, dass die betreffende Methode nie aufgerufen wird. In diesem Fall wird die Methode in der Implementation einer Basisklasse aufgerufen. Wenn eine Methode tatsächlich nie aufgerufen wird und somit toten Code darstellt, wird in jedem Falle explizit darauf hingewiesen.

(1) Klasse TRadicon



Diese Klasse hat die Aufgabe, die Funktionalität des Detektors *Radicon SCSCS* gegenüber XCTL zu kapseln. Entwicklungsgeschichtlich hat es sich

ergeben, dass hier lediglich ein high-level-Interface zum Detektor implementiert wurde. Die eigentlichen Hardwarezugriffe werden in der Klasse *TRadiconHW* realisiert. Aus diesem Grund sind hier nur geringfügige Portierungsanpassungen erforderlich.

(2) Klasse TRadiconHW

Die Klasse *TRadiconHW* kapselt die Kommunikation mit der Detektorsteuerkarte vom Typ *Radicon*. Auch diese Steuerkarte verwendet Port-I/O für die Hardwarekommunikation. Die Durchführung dieser Kommunikation erfolgt mit Hilfe der Klasse *TIOPort*.

private Attribute		
Typ	Attribut	Beschreibung
TIOPort	ControlPort	Kommunikationsobjekt für den Kontrollport
TIOPort	DataPort	Kommunikationsobjekt für den Datenport

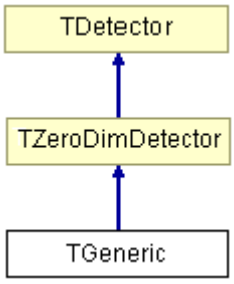
öffentliche Methoden	
TRadiconHW (int controlPortParam, int dataPortParam)	
verwendet wird	-
Beschreibung	Konstruktor
int **UploadFirmware** (void)	
verwendet wird	Execute(), ::GetDirectory(), in_byte(), TransmitMessage()
wird verwendet von	TRadicon::Initialize()
Beschreibung	dient dem Hochladen der Radiconfirmware in der Datei *scs.prg*
int **SetParameters** (unsigned short upperThreshold, unsigned short lowerThreshold, int Highvoltage, double exposureTime, unsigned long impulseCount, BOOL bSound)	
verwendet wird	TransmitMessage()
wird verwendet von	TRadicon::_SetParameters()
Beschreibung	setzt Parameterwerte
int **GetValues** (EReadoutMode, double *exposureTime, unsigned long *impulseCount)	
verwendet wird	ReceiveMessage()
wird verwendet von	TRadicon::EventHandler(), TRadicon::PollDetector()
Beschreibung	ermittelt Parameterwerte
int **Execute** (EOperationMode mode)	
verwendet wird	TransmitMessage()
wird verwendet von	TRadicon::EventHandler(), TRadicon::InitializeEvent(), TRadicon::MeasureStart(), TRadicon::MeasureStop(), TRadicon::PollDetector(), UploadFirmware()

Beschreibung	ruft *TransmitMessage()* in abhängigkeit von *mode* auf
void **reset** ()	
verwendet wird	TIOPort::Out8Bit() ControlPort
wird verwendet von	TRadicon::Initialize()
Beschreibung	führt einen Reset des Controllers aus

private Methoden	
int **TransmitMessage** (int nFunctionCode, unsigned char *lpszMessage=0, int nMessageLength=0)	
verwendet wird	in_byte(), out_byte(), TransmitFunctionCode()
wird verwendet von	Execute(), SetParameters() und UploadFirmware()
Beschreibung	übermittelt eine Nachricht an den Controller
int **ReceiveMessage** (int nFunctionCode, unsigned char *lpszMessage, int nMaxMessageLength, int &nRealMessageLength)	
verwendet wird	in_byte(), out_byte(), TransmitFunctionCode()
wird verwendet von	GetValues()
Beschreibung	empfängt eine Nachricht vom Controller
int **TransmitFunctionCode** (int nFunctionCode)	
verwendet wird	ReceiveMessage(), TransmitMessage()
wird verwendet von	
Beschreibung	sendet Funktionscode an Controller und wertet dessen Antwort aus
BOOL **out_byte** (unsigned char d)	
verwendet wird	CurrentTime(), dataPort, IsReadyToWrite(), TIOPort::Out8Bit()
wird verwendet von	ReceiveMessage(), TransmitFunctionCode(), TransmitMessage()
Beschreibung	sendet ein Byte an den Controller
BOOL **in_byte** (unsigned char &d)	
verwendet wird	CurrentTime(), dataPort, TIOPort::In8Bit(), IsReadyToRead()
wird verwendet von	ReceiveMessage(), TransmitFunctionCode(), TransmitMessage(), UploadFirmware()
Beschreibung	empfängt ein Byte vom Controller
BOOL **IsReadyToRead** ()	
verwendet wird	TIOPort::In8Bit() ControlPort
wird verwendet von	in_byte()
Beschreibung	testet Controller auf Lesebereitschaft
BOOL **IsReadyToWrite** ()	
verwendet wird	TIOPort::In8Bit() ControlPort
wird verwendet von	out_byte()
Beschreibung	testet Controller auf Schreibbereitschaft
DWORD **CurrentTime** (void)	
verwendet wird	
wird verwendet von	in_byte(), out_byte()

Beschreibung	liefert die aktuelle Zeit in Sekunden, seit das System gestartet wurde

(3) Klasse TGenericDetector



Diese Klasse hat die Aufgabe, die Funktionalität des Detektors *Generic SCSCS* gegebüber XCTL zu kapseln. Hierfür implementiert sie lediglich ein high-level-Interface zum Detektor. Die eigentlichen Hardwarezugriffe werden in der Klasse *TAm9513* realisiert. Aus diesem Grund sind hier nur geringfügige Portierungsanpassungen erforderlich.

(4) Klasse TAm9513

Aufgabe dieser Klasse ist die Kapselung der Kommunikation mit der Zählerkarte *AX5216* für die Detektorklasse *TGenericDetector*. Die Bezeichnung dieser Klasse ist vom verwendeten Timingcontroller mit der Bezeichnung *TAm9513* abgeleitet. Es handelt sich bei dieser Erweiterungskarte um keine echte Detektorkarte, sondern um eine universelle Zählerkarte, welche die Impulse eines externen Messgerätes zählt. Aus diesem Grund weicht die Implementation dieser Klasse zum Teil von den anderen Detektorklassen ab.

geschützte Attribute		
Typ	Attribut	Beschreibung
BYTE	**nbChipId**	Bedeutung unklar, wird genau einmal gesetzt (auf 0) und genau einmal ausgewertet (== 0x02), also quasi toter Code

private Attribute		
Typ	Attribut	Beschreibung
Int	nBaseAddr	Basisadresse des Controllers
Float	fTimeCorrection	Korrekturwert

BOOL	bIsLatched	Status
BOOL	bIsStopped	Status
WORD	wLowTicks	niederwertiges Wort Ticks
WORD	wHighTicks	höherwertiges Wort Ticks
WORD	wLowCounts	niederwertiges Wort Zähler
WORD	wHighCounts	höherwertiges Wort Zähler

öffentliche Methoden	
TAm9513 (void)	
verwendet wird	bIsLatched, bIsStopped, nbChipId, wHighCounts, wHighTicks, wLowCounts , wLowTicks
Beschreibung	Konstruktor
int **Init** (void)	
verwendet wird	ChooseDataPtr(), ReadData(), Reset(), WriteCmd(), WriteData()
wird verwendet von	TGenericDetector::Initialize()
Beschreibung	initialisiert die Zählerkarte
int **IOCTL** (TIOCCmd, DWORD &)	
verwendet wird	ArmC(), bIsLatched, bIsStopped, ChooseDataPtr(), ClearToggleOut(), ::DelayTime(), DisarmC(), fTimeCorrection, LatchToHoldC(), LoadAndArmC(), LoadC(), ReadData(), ReadStatus(), SetToggleOut(), SplitNumber(), wHighCounts, wHighTicks, wLowCounts, wLowTicks, WriteData()
wird verwendet von	TGenericDetector::_SetParameters(), TGenericDetector::Initialize(), IOCTL(), TGenericDetector::MeasureStart(), TGenericDetector::MeasureStop(), TGenericDetector::PollDetector()
Beschreibung	generiert Befehle für die Zählerkarte
int **IOCTL** (TIOCCmd)	
verwendet wird	IOCTL()
wird verwendet von	
Beschreibung	generiert Befehle für die Zählerkarte, vereinfachte Parameterliste
void **SetSound**(BOOL)	
verwendet wird	Controller::Hardware, HardwareIo::Write()
wird verwendet von	TGenericDetector::_SetParameters()
Beschreibung	Bedeutung unklar, da der TAm9513 keinen Lautsprecher hat
DWORD **GetTicksPerSecond** (void)	
verwendet wird	
wird verwendet von	TGenericDetector::_SetParameters() , TGenericDetector::PollDetector()
Beschreibung	liefert immer 100000 zurück, sonst keine Funktionalität
void **LookUp** (LPCSTR)	
verwendet wird	::GetHWFile() nBaseAddr, fTimeCorrection
wird verwendet von	TGenericDetector::TGenericDetector()

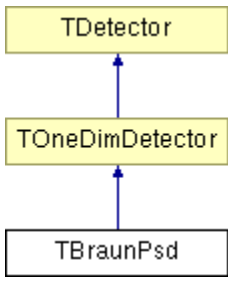
Beschreibung	liest Parameter aus der Datei *hardware.ini*
void **ClearToggleOut** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Bedeutung unklar
void **SetToggleOut** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Bedeutung unklar

geschützte Methoden	
BOOL **SelectChip** ()	
verwendet wird	ChooseDataPtr(), Delay(), ReadData(), Reset(), WriteData()
wird verwendet von	
Beschreibung	Bedeutung unklar
void **WriteCmd** (BYTE cmd)	
verwendet wird	::Delay() nBaseAddr, nbChipId
wird verwendet von	ArmC(), ChooseDataPtr(), ClearToggleOut(), DisarmAndSaveC(), DisarmC(), Init(), LatchToHoldC(), LoadAndArmC(), LoadC(), Reset(), SetToggleOut()
Beschreibung	sendet Befehl an Zählerkarte
void **ChooseDataPtr** (BYTE group, BYTE reg)	
verwendet wird	WriteCmd()
wird verwendet von	Init(), IOCTL(), SelectChip()
Beschreibung	dient der Adressierung von Registern auf der Zählerkarte
void **WriteData** (WORD)	
verwendet wird	::Delay() nBaseAddr
wird verwendet von	Init(), IOCTL(), SelectChip()
Beschreibung	schreibt ein WORD ins Datenregister
WORD **ReadData** (void)	
verwendet wird	::Delay() nBaseAddr
wird verwendet von	Init(), IOCTL(), SelectChip()
Beschreibung	liest ein WORD aus dem Datenregister
WORD **ReadStatus** (void)	
verwendet wird	::Delay() nBaseAddr
wird verwendet von	IOCTL()
Beschreibung	ermittelt den Status der Zählerkarte
void **LoadAndArmC** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Zählersteuerung

void **LoadC** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Zählersteuerung
void **DisarmAndSaveC** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	
Beschreibung	Zählersteuerung
void **DisarmC** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Zählersteuerung
void **LatchToHoldC** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Zählersteuerung
void **ArmC** (BYTE)	
verwendet wird	WriteCmd()
wird verwendet von	IOCTL()
Beschreibung	Zählersteuerung
void **Reset** (void)	
verwendet wird	WriteCmd()
wird verwendet von	Init(), SelectChip()
Beschreibung	führt einen Reset der Zählerkarte durch

private Methoden	
BOOL **SplitNumber** (DWORD, WORD &, WORD &)	
verwendet wird	
wird verwendet von	IOCTL()
Beschreibung	aufteilen eines DWORD in ein WORD, der Algorithmus ist unklar

(5) Klasse TBraunPSD



Die Klasse *TBraunPSD* kapselt die Kommunikation mit der *ASA*-Steuerkarte des Detektors *Braun PSD 50M*. Auch diese Karte implementiert den Port-I/O-Mechanismus. Prinzipiell könnte die Hardwarekommunikation über die Bibliotheksfunktionen *inp()* bzw. *outp()* realisieren werden, die Klasse *TBraunPSD* greift hierfür jedoch auf die vom Hersteller der Karte gelieferte Funktionsbibliothek *ASA.DLL* zurück.

öffentliche Attribute		
Typ	Attribut	Beschreibung
Int	nErrorCode	enthält den aktuellen Fehlercode, dient als Index, um entsprechenden Fehlerstring aus TBraunError[ ] zu erhalten
BOOL	bSetError	dient dazu anzuzeigen, dass der Fehlercode in *nErrorCode* geändert wurde, wird häufig gesetzt, jedoch nie ausgewertet
Int	nHVRegelung_OK	Bedeutung unklar, wird nie verwendet

private Atribute		
Typ	Attribut	Beschreibung
Int	nBaseAddr	Basisadresse der Controllerkarte
UINT	uMuxTimeDet1	Bedeutung unklar
LONG	PositionsDatenHeader [16]	Bedeutung unklar, wird nie verwendet
UINT	uEnergyScale	Energieskalierung
UINT	uPositionScale	Positionsskalierung
UINT	uEnergyHigh	Obere Grenze für das Energiefenster
UINT	uEnergyLow	Untere Grenze für das Energiefenster
LONG	lPositionStop	Bedeutung unklar, wird nie verwendet
BOOL	bAbbruchmitShutter	legt fest, ob bei Abbruch der Messung der Shutter geschlossen werden soll
Int	nDeathTime	minimaler Impulsabstand
BOOL	bRatemeter	legt den Modus für das externe Ratemeter fest: computed=0, total=1
BOOL	bRealLifeTime	Realtime=00, Lifetime=01
LONG	lMeasTime	Messzeit
HGLOBAL	hReadBuf	Handle des Lesepuffers, der an *GetData()* der *ASA.DLL* übergeben wird
Int	nDelayFast	legt die Wartezeit für Zugriffe auf die Controllerkarte fest
Int	nDelaySlow	legt die Wartezeit für Zugriffe auf die Controllerkarte fest
UINT	uCBufferLength	Größe des Lesepuffers mit dem Handle *hReadBuf*
Int	nReadBufItems	Bedeutung unklar, wird nie verwendet
Int	nHVControl_OK	Bedeutung unklar, wird gesetzt, aber nie ausgewertet

statische private Attribute		
Typ	Attribut	Beschreibung
HINSTANCE	AsaDllInstance	Handle auf 16-bit ASA.DLL Instanz
BYTE	echo[30][16]	Array für die Echos der ASA-Karte auf Befehle
BYTE	befehl [30][16]	Array für den Befehlssatz der ASA-Karte

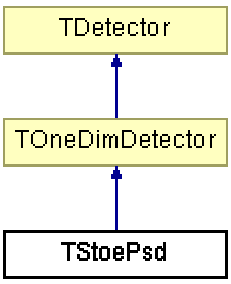
öffentliche Methoden	
TBraunPsd (int id)	
verwendet wird	AsaDllInstance, bRatemeter, bRealLifeTime, bSetError, GetBufferSize(), hReadBuf, LoadDetectorSettings(), nDeathTime, nErrorCode, TOneDimDetector::SetDataType, uEnergyHigh, uEnergyLow, uEnergyScale, uMuxTimeDet1, uPositionScale
Beschreibung	Konstruktor, unter anderem wird hier die ASA.DLL initialisiert und die Adressen der Funktionen, die aus ihr verwendet werden, werden den entsprechenden Zeigern zugewiesen
~**TBraunPsd** (void)	
verwendet wird	AsaDllInstance, bSetError, BuildOperation(), hReadBuf, nDelayFast, nErrorCode, SaveDetectorSettings()
Beschreibung	Destruktor
BOOL **Initialize** (void)	
verwendet wird	TOneDimDetector::Initialize()
wird verwendet von	
Beschreibung	initialisert den Detektor und prüft (wenn vorhanden), ob die Hardwarekommunikation mit dem Detektor möglich ist
int **PsdInit** (void)	
verwendet wird	bAbbruchmitShutter, bSetError, BuildOperation(), TOneDimDetector::GetHVControl(), LoadHexFile(), nDelayFast, nErrorCode, ResetDelayTime(), ::SetInfo()
wird verwendet von	
Beschreibung	initialisert den Detektor
int **PsdStart** (void)	
verwendet wird	bSetError, BuildOperation(), TOneDimDetector::eDataType, nDelayFast, nErrorCode, PsdEnergyData
wird verwendet von	
Beschreibung	dient dazu, die PSD-Hardware zum Start der Messung zu veranlassen
int **PsdReadOut** (THowReadOutPsd)	
verwendet wird	TOneDimDetector::bHardOverflow, bSetError, BuildOperation(), TOneDimDetector::CheckMaxChannel(), ::DelayTime(), TOneDimDetector::dwIntegratedCounts, TOneDimDetector::dwMaxCounts, TOneDimDetector::eDataType, GetChannelNumber(), TOneDimDetector::GetFirstChannel(), TOneDimDetector::GetHVControl(), TOneDimDetector::hCountBuf,

	MessDatenHeader::HighADCCounts, hReadBuf, MessDatenHeader::HVParameter, TOneDimDetector::nBaseAddr, nDelaySlow, nErrorCode, nHVControl_OK, ::SetStatus(), MessDatenHeader::Ueberlauf, uEnergyScale, TOneDimDetector::uMaximumChannel, uPositionScale
wird verwendet von	
Beschreibung	dient dazu, die Messwerte aus der PSD-Hardware zu lesen
int **GetBufferSize** (void)	
verwendet wird	uCBufferLength, uEnergyScale und uPositionScale
wird verwendet von	TBraunPsd()
Beschreibung	gibt die Größe des Datenpuffers für das Auslesen der Messwerte zurück
BOOL **SetEnergyRange** (UINT, UINT)	
verwendet wird	bSetError, BuildOperation(), nDelayFast, nErrorCode, uEnergyHigh, uEnergyLow, uEnergyScale
wird verwendet von	
Beschreibung	legt das zu verwendende Energiefenster fest
void **GetEnergyRange** (UINT &ler, UINT &her) const	
verwendet wird	uEnergyHigh, uEnergyLow
wird verwendet von	
Beschreibung	gibt das verwendete Energiefenster zurück
int **PsdStop** (void)	
verwendet wird	bAbbruchmitShutter, bSetError, BuildOperation(), nDelayFast, nErrorCode
wird verwendet von	
Beschreibung	dient dazu, die PSD-Hardware zum Stopp der Messung zu veranlassen
int **GetChannelNumber** () const	
verwendet wird	TOneDimDetector::eDataType, TOneDimDetector::GetChannelNumber() uEnergyScale
wird verwendet von	PsdReadOut()
Beschreibung	gibt die Anzahl der zu nutzenden Kanäle zurück
TDetectorType **GetDetectorType** () const	
verwendet wird	bSetError, konvert(), Look_till_BaseAddr1(), TOneDimDetector::nBaseAddr, nErrorCode, ::SetInfo(), SynchronHexFile()
wird verwendet von	PsdInit()
Beschreibung	gibt eine symbolische Konstante zurück, die den Typ des Detektors identifiziert

geschützte Methoden	
int **BuildOperation** (BYTE *, BYTE *, int)	
verwendet wird	::Delay(), Look_till_BaseAddr1(), TOneDimDetector::nBaseAddr, SetInfo()

wird verwendet von	PsdInit(), PsdReadOut(), PsdStart(), PsdStop(), SetEnergyRange(), ~TBraunPsd()
Beschreibung	sendet eine Befehlssequenz an den Controller und überprüft dessen Antwort auf diese Befehlssequenz
int **LoadHexFile** (void)	
verwendet wird	konvert(), Look_till_BaseAddr1(), TOneDimDetector::nBaseAddr, nErrorCode, :: SetInfo(), SynchronHexFile()
wird verwendet von	PsdInit()
Beschreibung	lädt die Datei *ASA23.hex* auf die Controllerkarte (Firmware)
void **ResetDelayTime** (void)	
verwendet wird	TOneDimDetector::nBaseAddr, SetInfo()
wird verwendet von	PsdInit()
Beschreibung	Bedeutung unklar
BYTE **konvert** (char)	
verwendet wird	
wird verwendet von	LoadHexFile()
Beschreibung	Bedeutung unklar
int **SynchronHexFile** (BYTE &, BYTE &)	
verwendet wird	Look_till_BaseAddr1(), TOneDimDetector::nBaseAddr, ::SetInfo()
wird verwendet von	LoadHexFile()
Beschreibung	Synchronisation mit dem Controller für das Hochladen der Datei *ASA23.hex*
int **Look_till_BaseAddr1** (int)	
verwendet wird	::Delay(), ::SetInfo() TOneDimDetector::nBaseAddr
wird verwendet von	BuildOperation(), LoadHexFile(), SynchronHexFile()
Beschreibung	wartet darauf, dass der Controller Bereitschaft signalisiert
void **LoadDetectorSettings** ()	
verwendet wird	TDetector::GetPrivateProfileString(), bAbbruchmitShutter, bRatemeter, bRealLifeTime,nDeathTime, nDelayFast, nDelaySlow, uEnergyHigh, uEnergyLow, uEnergyScale, uMuxTimeDet1, uPositionScale
wird verwendet von	TBraunPsd()
Beschreibung	lädt Einstellungen für den Detektor aus der Datei *hardware.ini*
void **SaveDetectorSettings** () const	
verwendet wird	uEnergyHigh, uEnergyLow, TDetector::WritePrivateProfileString()
wird verwendet von	~TBraunPsd()
Beschreibung	speichert Einstellungen für den Detektor in der Datei *hardware.ini*

(5) Klasse TStoePSD



Diese Klasse kapselt die Zugriffe auf den Detektor vom Typ *Stoe-PSD*. Die Hardwarezugriffsfunktionen sind zum Teil noch in inline-Assembler implementiert. Da der Detektor in ein proprietäres Messplatzsystem integriert ist, wurde er bislang nie mit XCTL verwendet. Es gibt auch vorläufig keine Pläne, diesen Detektor mit XCTL zu verwenden. Aus diesem Grund ist diese Klasse bei der Portierung als irrelevant anzusehen. Im aktuellen Zustand sind die kritischen Bereiche unter Win32 durch die Direktive *#ifndef _WIN32* deaktiviert.

(6) globale Variablen

Typ	Variable	Beschreibung
void (WINAPI *lpfnSetPort) (WORD, WORD);	lpfnSetPort	Funktionszeiger für die ASA.DLL Funktion SetPort()
BYTE (WINAPI *lpfnGetPort) (WORD);	lpfnGetPort	Funktionszeiger für die ASA.DLL Funktion GetPort()
void (WINAPI *lpfnSetTimeout) (DWORD);	lpfnSetTimeout	Funktionszeiger für die ASA.DLL Funktion SetTimeout ()
int (WINAPI *lpfnGetData) (WORD, WORD, WORD, LPLONG, WORD, long&);	lpfnGetData	Funktionszeiger für die ASA.DLL Funktion GetData ()

Analyse – Hwio.dll

(1) Klasse TIOPort

Diese Klasse kapselt Hardwarezugriffsfunktionen für den Port-I/O-Mechanismus in Abhängigkeit von der eingesetzten Windowsversion. Semantisch verkörpert ein Objekt dieser Klasse einen I/O-Port, welcher mit den Methoden der Klasse gelesen oder geschrieben werden kann.

Die Umsetzung der Hardwarekommunikation wird über die Direktive *#ifdef _WIN32* gesteuert. Wird der Quelltext unter Win16 (Windows 3.11) übersetzt, werden intern die üblichen Portzugriffsfunktionen (*inp()*/*outp()*) genutzt. Bei der Erstellung des Moduls unter Win32 (Windows 2000) war es beabsichtigt, die Hardwarekommunikation über einen im System installierten generischen Porttreiber zu realisieren. Diese Umsetzung wurde aber nicht abschließend implementiert, so dass die _WIN32-Zweige nicht die korrekte Funktionalität zur Verfügung stellen.

private Attribute		
Typ	Attribut	Beschreibung
unsigned int	port	Portadresse des Portobjektes

öffentliche Methoden	
TIOPort (unsigned int p)	
verwendet wird	
Beschreibung	Konstruktor
BYTE **In8Bit** (void)	
verwendet wird	port
wird verwendet von	TRadiconHW::in_byte(), TRadiconHW::IsReadyToRead(), TRadiconHW::IsReadyToWrite(), TRadiconHW::TransmitFunctionCode()
Beschreibung	schreibt den 8-bit Port, den das Objekt verkörpert
WORD **In16Bit** (void)	
verwendet wird	port
wird verwendet von	TStoePsd::PsdStop()
Beschreibung	schreibt den 16-bit Port, den das Objekt verkörpert
void **Out8Bit** (BYTE d)	
verwendet wird	port
wird verwendet von	TRadiconHW::out_byte(), TRadiconHW::reset()
Beschreibung	liest den 8-bit Port, den das Objekt verkörpert
void **Out16Bit** (WORD d)	
verwendet wird	port
wird verwendet von	TStoePsd::PsdStart(), TStoePsd::PsdStop()
Beschreibung	liest den 16-bit Port, den das Objekt verkörpert

Zusätzliche portierungsrelevante Funktionen

Die hier aufgeführten Funktionen sind im Modul *Splib.dll* implementiert und stehen nur indirekt mit der Hardwaresteuerung in Verbindung. Bei

diesen Funktionen handelt es sich im Wesentlichen um „Wartefunktionen“, die der Steuerung des zeitlichen Ablaufs der Hardwarekommunikation dienen. Da diese Kommunikation stark zeitabhängig ist, zeigte sich im Laufe der Portierung, dass diese Quelltextfragmente einen starken Einfluss auf die korrekte Funktion der Hardwareansteuerung ausüben. Falsche Wartezeiten führen zu Fehlfunktionen.
Leider sind die Funktionen hardwareabhängig implementiert, so dass die unterschiedlichen Ausführungsgeschwindigkeiten verschiedener Rechnerplattformen zu einem undefinierten Zeitverhalten der Funktionen führen. Dieses Verhalten muss im Zuge der Portierung korrigiert werden.

void WINAPI _export **Delay**(long count)	
verwendet wird	
wird verwendet von	TBraunPsd::BuildOperation(), TScan::Create(), TAreaScan::Create(), DelayTime(), TCalibratePsdDlg::Dlg_OnInit(), TC_812ISA::ExecuteCmd(), TC_812GPIB::ExecuteCmd(), TC_812ISA::GetChar(), TC_832::IsIndexArrived(), TBraunPsd::Look_till_BaseAddr1(), TStoePsd::PsdInit(), TStoePsd::PsdRead(), TStoePsd::PsdStart(), TStoePsd::PsdStop(), TC_812ISA::PutChar(), PutWord(), TAm9513::ReadData(), TAm9513::ReadStatus(), TAm9513::SelectChip(), TC_812::StopDrive(), TAm9513::WriteCmd(), TAm9513::WriteData()
Beschreibung	wartet eine bestimmte Zeit, unklar ist, welche Einheit gemeint ist oder wie lang eine Einheit ist, definitiv ist diese Funktion extrem abhängig von der Leistung der unterliegenden Hardware, da diese Funktion durch eine Schleife realisiert ist

void WINAPI _export **DelayTime**(int MilliSec)	
verwendet wird	Delay()
wird verwendet von	TRadicon::_SetParameters(), TC_812ISA::CheckBoardOk(), TTopographyExecuteDlg::Dlg_OnTimer(), TAngleCtl::DoStop(), TAm9513::IOCTL(), TC_812::IsIndexArrived(), TC_832::IsLimitHit(), TC_832::IsMoveFinish(), TC_812ISA::IsMoveFinish(), TTopographyExecDlg::OnTimer(), TOneDimDetector::PollDetector(), TStoePsd::PollDetector(), TRadicon::PollDetector(), TBraunPsd::PsdReadOut(), TCmd::StartMove(), TC_832::StartToIndex(), TC_812::StartToIndex(), TC_832::StopDrive(), TCmd::WakeUp()
Beschreibung	wartet eine bestimmte Zeit in Millisekunden

Analyse – Win488.dll

Da die DLL nicht im Quelltext vorliegt, kann für eine Analyse der implementierten Funktionalität nur von der Art der Nutzung dieser DLL durch XCTL ausgegegangen werden. Der einzige Nutzer der DLL ist die Klasse *TC812GPIB*, so dass als relativ sicher angesehen werden kann, dass sich die von XCTL genutzte Funktionalität ausschließlich auf Hardwarekommunikation nach dem IEEE-488-Standard beschränkt. Da diese Art der Hardwareansteuerung durch das Zielsystem nicht mehr unterstützt werden muss, kann von einer weiteren Analyse und entsprechenden Adaptionsschritten abgesehen werden.

Analyse – Asa.dll

Hier liegen die Quelltexte ebenfalls nicht vor, so dass ein äquivalentes Vorgehen für die Analyse der Funktionalität notwendig ist. Die Analyse zeigt, dass XCTL die schon in Tabelle [5.1] aufgeführten Funktionen der DLL nutzt.
Da es sich um eine 16-bit-DLL handelt, die somit nicht unter Win32 ausgeführt werden kann, wird die entsprechende Funktionalität im Gerätetreiber für den Detektor *Braun-PSD* reimplementiert.

Zäsur

Die Ausgangssituation ist nun hinreichend geklärt und die potenziell zu ändernden Quelltextteile sind identifiziert, so dass die notwendigen Transformationen vorgenommen werden können. Was ist in diesem Rahmen zu tun?

(1) Spezifikation und Erstellung der XCTL-Gerätetreiber.

(2) Umstellung der direkten Hardwarezugriffe von XCTL auf Treiberzugriffe.

Spezifikation der XCTL-Gerätetreiber

(1) Überblick
Um *XCTL* nach Windows 2000 portieren zu können, bedarf es einer Möglichkeit, auf die anwendungsspezifische Hardware zugreifen zu können. Dies ist durch den Einsatz Windows-2000-konformer Gerätetreiber zu erreichen.

Der Funktionsumfang eines Treibers für *XCTL* lässt sich in zwei Kategorien einteilen:

1. Funktionen, die vom Betriebssystem zur Verwaltung von Treiber und Geräten verlangt werden.

2. Funktionen, die den Datenaustausch mit der Hardware bewerkstelligen.

    a. Grundfunktionalität - Byte-weiser Datentransfer von und zur Hardware.

    b. erweiterte Funktionalität - Verlagerung von Funktionalität, die bislang in XCTL implementiert war, in den Treiber

Die im Rahmen dieser Portierung entwickelten Gerätetreiber implementieren die Verwaltungs- und Grundfunktionalität.

(2) Funktionale Beschreibung
Bei den zu erstellenden Gerätetreibern handet es sich um Funktionstreiber(vergl. Tabelle [3.2]). Sie haben die Aufgabe, die XCTL–Hardware zu verwalten und eine operationale Schnittstelle (via IOCTL-Codes) dieser Geräte in das Betriebssystem zu exportieren (vergl. Punkt 2.1 der Treiberspezifikation).
Um einen Gerätetreiber in das System integrieren und diesen dann dort verwalten zu können, sind sogenannte *Installer* erforderlich. Dabei handelt es sich um auf Installation und Verwaltung bestimmter Hardware spezialisierte Softwaremodule (DLLs) (vergl. Punkt 2.2 der Treiberspezifikation).

(2.1) Funktionstreiber
Für jedes der zu verwaltenden Geräte (vergl. Tabellen [3.5.1], [3.5.2]) wird ein eigener Gerätetreiber benötigt. Die Bezeichnungen der Treiber und die

dem jeweiligen Treiber zugeordnete XCTL-Hardware kann der Tabelle [5.7] (siehe Treiberspezifikation Abschnitt 4.1) entnommen werden. Des weiteren können Treiber ohne Hardwarezugriffe für Entwicklungszwecke erstellt werden.

(2.1.1) Verwaltungsfunktionen
Als Verwaltungsfunktionen werden sämtliche Funktionen bezeichnet, die dafür verantwortlich sind, den Treiber zu initialisieren und die Kommunikation mit dem E/A-Manager und dem PnP-Manager des Betriebssystems auszuführen. Die Erstellung eines Funktionstreibers für Windows 2000 erfordert die Implementation von Verwaltungsfunktionen (Tabelle [5.5]). Näheres zu den Anforderungen an diese Verwaltungsfunktionen ist unter [Oney03], [MSDN03b], [Solomon00] zu finden.

Die Initialisierung eines Treibers soll zusätzlich folgende Funktionalität umfassen:

1. Überprüfung, ob die zugewiesenen Ressourcen den Hardwareanforderungen entsprechen (siehe Tabelle [D.1])
2. Test der Kommunikation mit dem zu verwaltenden Gerät
3. Sobald einer der genannten Vorgänge nicht erfolgreich abgeschlossen werden konnte, ist dies dem System mitzuteilen, so dass der Treiber sofort wieder entladen wird. Damit ist sichergestellt, dass nur erfolgreich initialisierte Treiber dem XCTL-System zur Verfügung stehen.

Verwaltungsfunktionen für Funktionstreiber	
Funktion	**Beschreibung**
DriverEntry()	Treibereinstiegspunkt, initialisiert den Treiber, initialisiert die Einbindung des Treibers ins E/A-System
XxAddDevice()	Erzeugung funktionaler Geräteobjekte (Anlegen einer neuen Treiberinstanz je physischem Gerät)
XxDispatchPNP()	bedient Kommunikationsanforderungen des PnP Managers (Neben-IRPs von IRP_MJ_PNP)
XxDispatchCreateClose()	bedient Kommunikationsanforderungen des E/A-Managers (IRP_MJ_CREATE und IRP_MJ_CLOSE; gesendet, wenn Benutzermodusprogramm ein Handle auf den Treiber anlegt/löscht)
XxUnload()	wird vor dem Entladen des Treibers aufgerufen
XxDispatchDeviceControl()	bedient Kommunikationsanforderungen des E/A-Managers (IOCTL-Codes des IRP IRP_MJ_DEVICE_CONTROL; angefordert via

	DeviceIoControl() aus dem Benutzermodus)

Tabelle [5.5]
Anmerkung: eine vollständige Treiberbeschreibung befindet sich in Anhang A

(2.1.2) Funktionen für den Datenaustausch mit der Hardware
Neben den Verwaltungsfunktionen wird eine Funktionalität benötigt, welche *XCTL* den Zugriff auf die Hardware ermöglicht. Dies wird in Form des IOCTL-Mechanismus umgesetzt. Die folgende Tabelle [5.6] beschreibt die Funktionscodes der IOCTL-Schnittstelle eines XCTL-Treibers.
Die Funktionen der IOCTL-Schnittstelle dürfen nur im Bereich der dem Treiber zugewiesenen Ressourcen operieren. Anforderungen über die Schnittstelle sind dahingehend zu prüfen und ggf. zurückzuweisen. Die Adressierung der entsprechenden I/O-Ports bzw. Speicherstellen erfolgt über Offsets bezüglich der für den Treiber konfigurierten Basisadresse. Neben der IOCTL-Schnittstelle muss eine Funktion implementiert werden, die in der Lage ist zu erkennen, ob mit der unterstützten Hardware auch tatsächlich über die konfigurierten Ressourcen kommuniziert werden kann (Hardwareerkennung; siehe Verwaltungsfunktionen).

IOCTL-Interface der XCTL-Treiber	
ICOTL-Code	**Bedeutung**
IOCTL_DC_REPORT_ID	liefert die Kennung der aktuellen Treiberinstanz, entspricht der Basisadresse des von dieser Treiberinstanz verwalteten Geräts
IOCTL_DC_READ_BYTE	liest ein Byte vom angegebenen Offset (ab Basisadresse)
IOCTL_DC_WRITE_BYTE	schreibt ein Byte an den angegebenen Offset (ab Basisadresse)
IOCTL_DC_SET_WFR_CYCLES *)	*Set Wait For Ready Cycles* --> ersetzt SetTimeOut() Funktion der 16-bit ASA.DLL
IOCTL_DC_GET_DATA *)	Get Data --> ersetzt GetData() Funktion der 16-bit ASA.DLL

*) Diese IOCTL-Codes sind nur für den Gerätetreiber des Braun PSD vorgesehen, da dieser die 16-bit ASA.DLL ersetzen soll und somit mehr Funktionalität aufweisen muss als die anderen Gerätetreiber.
Tabelle [5.6]

(2.1.3) Treiberentwicklung
Um die Entwicklung der Gerätetreiber und des XCTL-Systems zu erleichtern, sollten Möglichkeiten zur Überwachung des Zustands eines Treibers vorgesehen werden.

(2.1.3.1) Debugfunktionalität für Entwicklungszwecke
Um den aktuellen Zustand der Treiber in der Entwicklungsphase überwachen zu können, sollten Ausgaben an einen Kernelmodedebugger [Anhang C – Werkzeuge, DebugView, WinDbg] gesendet werden. Diese sind so zu implementieren, dass sie nur im Debugmodus zur Verfügung stehen, da sie im realen Einsatz unnötig Ressourcen beanspruchen.

(2.1.3.2) Testtreiber für Entwicklungszwecke
Testtreiber[5] ohne echte Hardwarerepräsentation können entwickelt werden, um die Kommunikation zwischen XCTL und Treiber zu überwachen. Dies ist beispielsweise während der Treiberentwicklung von Vorteil, da echte Hardwaretreiber ggf. einen Neustart des Systems erfordern, Testtreiber jedoch nicht. Ein Testtreiber protokolliert die Kommunikation zwischen XCTL und Hardware in Log-Dateien.
Weiterhin sollen die Testtreiber als Grundlage für eine eventuell später erfolgende Verlagerung von Simulationsfunktionalität in die Treiber dienen.

(2.2) Installer
Um die Gerätetreiber im System zu integrieren und diese dann dort zu verwalten, werden vom Betriebssystem Module (DLL) gefordert, die diese Aufgaben übernehmen. Zusätzlich ist für den eigentlichen Installationsvorgang des Treibers ein Installationsskript erforderlich, welches folgende Aufgaben übernehmen muss:

(1) Kopieren aller erforderlichen Dateien an die entsprechenden Stellen im System
(2) Einbindung der Treiber in das System
(3) Übermittlung des Ressourcenbedarfs des zu installierenden Gerätes an das System bzw. die Abfrage dieser Ressourcen vom Nutzer
(4) Initialisieren der benötigten Registrierungsschlüssel
(5) Information des Nutzers über die installierbaren Treiber
(6) Einbindung des Installermoduls in das System

[5] Die Testtreiber sind als Nebenprodukt bei der Portierung entstanden und für die eigentliche Portierung nicht noptwendig. Daher ergibt sich keine explizite Forderung nach einer Spezifikation und separaten Tests.

(2.2.1) Klasseninstallerfunktionalität
Die XCTL-Hardware bildet zwei Gruppen von Geräten: Detektoren und Motoren. Je Gruppe soll eine eigene Geräteklasse angelegt werden:

XCTL Geräteklassenübersicht	
Gruppe	**Geräteklasse**
Detektoren	CDetectorControl
Motoren	CMotorControl

Die Klassenfunktionalität soll sich aktuell darauf beschränken, ein Symbol der Geräteklasse zur Verfügung zu stellen. Damit soll die Ergonomie der Benutzerinteraktion verbessert werden.

(2.2.2) Co-Installerfunktionalität
Ein Co-Installer kapselt die Installations- und Verwaltungsfunktionen des zu installierenden bzw. zu verwaltenden Gerätes. Die Co-Installerfunktionalität wird hier nur für die Testtreiber benötigt und stellt dort spezifische Dialogseiten des Installationsassistenten sowie eine spezifische Ressourceneigenschaftsseite (*Property Page Dialog*) des Gerätemanagers zur Verfügung. Hiermit soll der imaginäre Ressourcenbedarf eines Testtreibers vom Benutzer erfragt und in der Registry gespeichert werden. Der Treiber muss von den Änderungen in Kenntnis gesetzt werden, um einen Neustart des Systems zu vermeiden.

(3) Benutzerschnittstelle
Da Gerätetreiber autark operierende Softwaremodule sind, findet keine direkte Benutzerinteraktion statt. Lediglich während der Installation und der Verwaltung interagiert der Benutzer indirekt über das Betriebssystem in Form von *Installationsassistent* bzw. *Gerätemanager* mit dem Treiber.

(3.1) Installation
Während des Installationsvorgangs soll für die XCTL-Hardwaretreiber der systemeigene Ressourcendialog zur Abfrage der aktuellen Konfiguration verwendet werden.
Bei den Testtreibern kann dieser systemeigene Dialog nicht genutzt werden, da nicht wirklich Ressourcen vom System angefordert werden. Es muss also ein spezieller Dialog erzeugt und in den Installationsvorgang eingebracht werden. Über diesen werden dann die virtuell genutzten Ressourcen konfiguriert.

(3.2) Geräteverwaltung
Zur Verwaltung der Konfiguration der XCTL-Hardwaretreiber wird der systemeigene Ressourcendialog genutzt. Dieser ist in die vorgegebene Dialogstruktur des Gerätemanagers einzubinden.
Für die Testtreiber muss auch hier (vergl. 3.1) ein spezieller Dialog erzeugt und in die Dialogstruktur des Gerätemanagers eingebracht werden. Über diesen können dann die imaginären Ressourcen verändert werden.

(4) Dateien
Treiberfunktionalität und Funktionalität der Verwaltungsstrukturen werden in verschiedenen Dateien implementiert.

Treiberdateien	
Treiber	**Gerät**
MC812.SYS	Motorsteuerkarte C-812
MC832.SYS	Motorsteuerkarte C-812
DCGeneric.SYS	generische Zählerkarte AX5216
DCRadicon.SYS	Detektorkarte Radicon SCSCS
DCBraunPSD.SYS	ASA Detektorkarte des Braun-PSD

Tabelle [5.7.1]

Installer	
Intaller	**Geräteklasse**
MC8x2.DLL	CMotorControl
DCx.DLL	CDetectorControl

Tabelle [5.7.2]

Installationsskripte	
Installationsskript	**Geräteklasse**
MC8x2.INF	CMotorControl
DCx.INF	CDetectorControl

Tabelle [5.7.3]

(5) Sonstige Anforderungen
Um verschiedene Entwicklungsstadien von Treibern, Installermodulen und Installationsskript unterscheiden zu können, ist eine Versionierung vorzusehen, die sich an die Standards von Windows 2000 hält. Dies bedeutet im Einzelnen:

- für Treiber und Installer:
  Der Versionsstring muss über den Eigenschaftsdialog von *Windows-Explorer* und, im Falle der Treiber, *Gerätemanager* verfügbar sein

- für das Installationsskript:
  Die Version muss an entsprechender Stelle in der INF-Datei vermerkt werden [MSDN03b].

Die Art des Treibers (Hardwaretreiber bzw. Testtreiber) soll über den Eigenschaftdialog des Windows-Explorers ermittelbar sein.

Anmerkung: Eine vollständige Treiberbeschreibung befindet sich in Anhang A.

Umstellung der direkten Hardwarezugriffe auf Treiberzugriffe

In diesem Abschnitt werden die durchgeführten Anpassungen von *XCTL* dokumentiert und in ihrer Motivation erläutert. Dabei wird sowohl auf die Modifikationen existierender Quelltextteile als auch die Neuimplementation von Klassen eingegangen.

Bei den für XCTL entwickelten Gerätetreibern handelt es sich um s.g. WDM-Funktionstreiber. Ein solcher Gerätetreiber ist in der Lage, je Treiberinstanz den Zugriff auf ein physisches Gerät zu ermöglichen. Als *ein Gerät* zählt bei der XCTL-Hardware jede einzelne Controllerkarte und nicht die dort angeschlossene Hardware. Da die Detektorsteuerkarten jeweils nur einen Detektor kontrollieren, kann hier auch der jeweilige Detektor als physisches Gerät verstanden werden. Anders bei den Motorsteuerkarten. Hier steuert eine Karte gleich mehrere angeschlossene Motoren. Die einzelnen Motoren können also nicht individuell als physische Geräte angesehen werden. Die bisher in *XCTL* implementierten Hardwareklassen abstrahieren allerdings von physischen Geräten und bilden dementsprechend auch jeden Motor als ein solches Gerät ab. Um diesen Widerspruch aufzuheben, wird eine neue Abstraktionsschicht in *XCTL* eingeführt. Diese Schicht repräsentiert die tatsächlichen physischen Geräte und steht zwischen den bisherigen Hardwareklassen und der Treiberschicht (Hardware), so dass die Schnittstelle zwischen Motoren bzw. Detektoren und XCTL nicht verändert werden muss. Die neue Abstraktionsebene wird im Folgenden als *Controllerschicht* bezeichnet. Ihre wesentliche Aufgabe liegt in der Kapselung der Treiberzugriffe und Treiberverwaltung. Die folgende Abb. [5.2] zeigt die neue Architektur von *XCTL*.

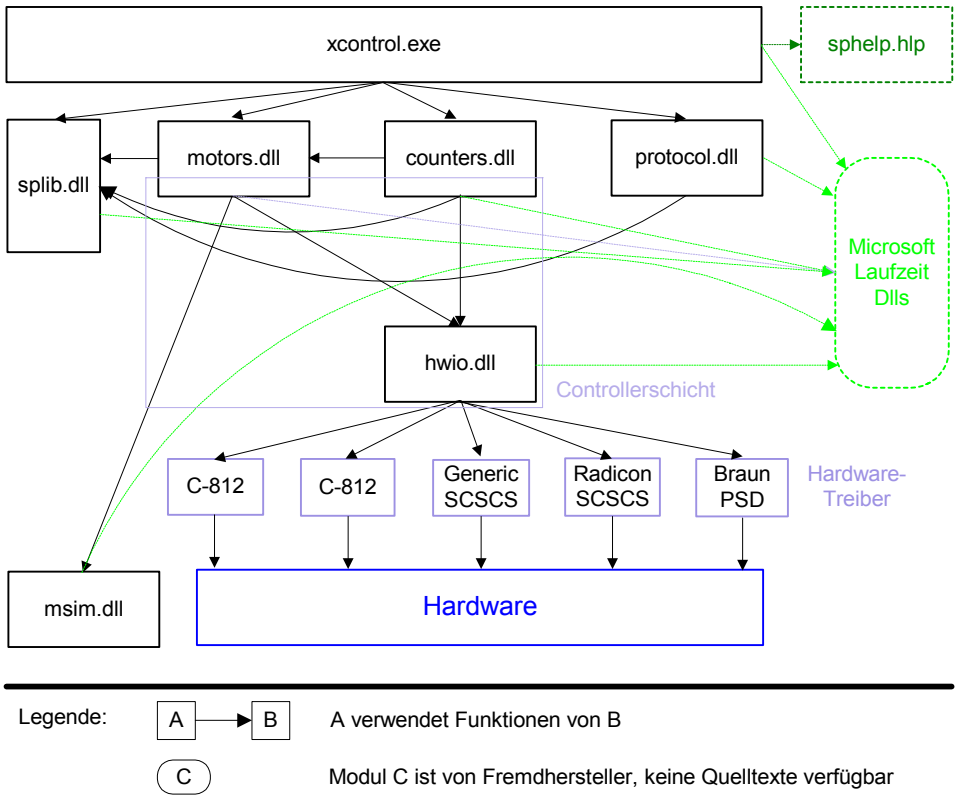


Abb. [5.2] Architektur XCTL (neu)

Die notwendigen Adaptionen bestehen nun in der Implementation der Controllerschicht und der Anpassung der Hardwareklassen, so dass diese die Funktionen der Controllerschicht an Stelle der direkten Hardwarezugriffe nutzen.

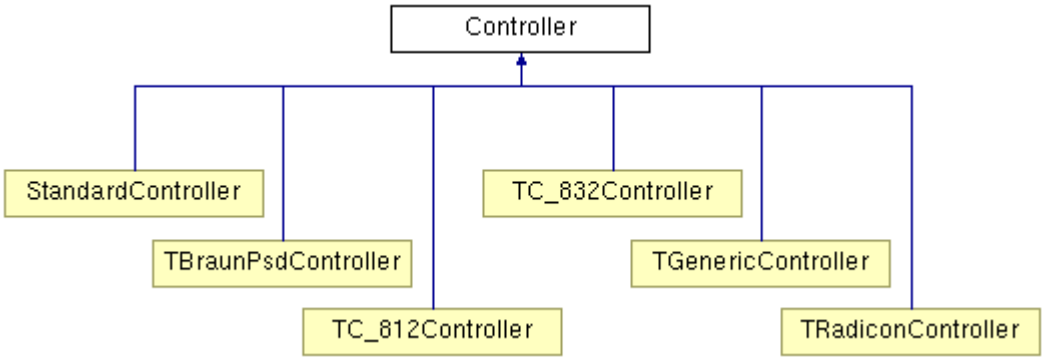
Klassen der Controllerschicht			
Klassenname	**Parent**	**Modul**	**Funktion**
DeviceList	-	Hardwarezugriff	Zuordnung von HardwareID des Gerätes und zugehörigem Treiber
DeviceListEntry	-	Hardwarezugriff	interne Datenstruktur von DeviceList
HardwareIo	-	Hardwarezugriff	Abstraktion des Treiber/Hardwarezugriffs
DummyIo	HardwareIo	Hardwarezugriff	Standardhandler für Hardware/ Treiberzugriffe
BraunPsdIo	HardwareIo	Hardwarezugriff	Abstraktion des Treiber/Hardwarezugriffs (Spezialisierung für Braun-PSD)

BraunPsdDummyIo	BraunPsdIo	Hardwarezugriff	Standardhandler für Hardware/ Treiberzugriffe (Spezialisierung für Braun-PSD)
Controller	-	Hardwarezugriff	Abstraktion eines physischen Gerätes (abstrakte Klasse)
ControllerList	-	Hardwarezugriff	Verwaltung der instantiierten Controller
ControllerListEntry	-	Hardwarezugriff	interne Datenstruktur von ControllerList
StandardController	Controller	Hardwarezugriff	unspezifischer Controller
TC812Controller	Controller	Motorsteuerung	Abstraktion Motorcontoller C-812
TC823Controller	Controller	Motorsteuerung	Abstraktion Motorcontoller C-832
TRadiconController	Controller	Detektornutzung	Abstraktion Detektor Radicon SCSCS
TGenericController	Controller	Detektornutzung	Abstraktion Detektor Generic SCSCS
TBraunPsdController	Controller	Detektornutzung	Abstraktion Detektor Braun-PSD

Tabelle [5.8]

Die neu erstellten Klassen (Tabelle [5.8]) der Controllerschicht werden im Folgenden beschrieben.

(1) Klasse Controller



Diese Klasse implementiert ein abstraktes Interface zu einem physischen Hardwaregerät und ist Basisklasse aller anderen Controllerklassen. Sämtliche Zugriffe von *XCTL* auf Detektoren und Motoren werden über das Interface geführt. Die genauen Mechanismen des Hardware- bzw. Treiberzugriffs werden hinter dieser Schnittstelle versteckt.

Controller
HardwareIo * **Hardware** # const DEVICETYPE **DeviceID** # char **HardwareID** [9] # unsigned **Clients** # const unsigned **MaxClients**
+ **Controller** (DEVICETYPE DeviceID, LPTSTR HardwareID, unsigned MaxClients) + virtual **~Controller** () + int **GetDeviceID** () + LPTSTR **GetHardwareID** () + BOOL **AddClient** () + virtual BOOL **Read** (DWORD Offset, BYTE *Data) + virtual BOOL **Read** (DWORD Offset, WORD *Data) + virtual BOOL **Read** (DWORD Offset, DWORD *Data) + virtual BOOL **Write** (DWORD Offset, BYTE Data) + virtual BOOL **Write** (DWORD Offset, WORD Data) + virtual BOOL **Write** (DWORD Offset, DWORD Data) + virtual BOOL **Check** ()=0

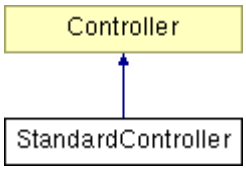
Die Klassen *ControllerList* und *ControllerListEntry* dienen lediglich der Verwaltung der *Controller*-Objekte (doppelt verkettete Liste).

ControllerListEntry
- Controller * **asController** - ControllerListEntry * **prev** - ControllerListEntry * **next**
+ **ControllerListEntry** (Controller *asController) + **~ControllerListEntry** () + void **SetPrevEntry** (ControllerListEntry *pEntry) + void **SetNextEntry** (ControllerListEntry *pEntry) + **ControllerListEntry** * GetPrevEntry () + **ControllerListEntry** * GetNextEntry () + int **GetDeviceID** () + LPTSTR **GetHardwareID** () + Controller * **GetController** ()

ControllerList
- ControllerListEntry * **first** - ControllerListEntry * **last**
+ **ControllerList** () + **~ControllerList** () + void **Add** (Controller *asController) + void **Add** (ControllerListEntry *pEntry) + void **Clear** () + Controller * **GetController** (int DeviceID, LPTSTR HardwareID)

+ Controller * **GetController** (unsigned index) + DWORD **GetControllerIndex** (int DeviceID, LPTSTR HardwareID)

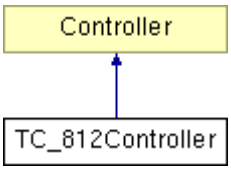
(2) Klasse StandardController



Die Klasse implementiert die Hardware/Treiberzugriffe für eine unspezifische Controllerkarte. Sie dient im Wesentlichen als generischer Controller für unbekannte Geräte und kann als Basis für ggf. später einzubindende Hardwaregeräte dienen.

StandardController
<>
+ **StandardController** (DEVICETYPE DeviceID, LPTSTR HardwareID, LPTSTR DeviceName) + **StandardController** (DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList *Devices) + virtual ~**StandardController** () + virtual **BOOL Check** ()

(3) Klasse TC_812Controller

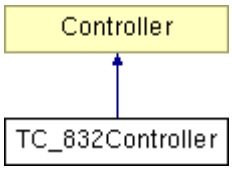


Die Klasse implementiert die Hardware/Treiberzugriffe für eine *C-812* Motorcontrollerkarte. Die globalen Funktionen *C812ISA_Get()* und *C812ISA_Put()* werden durch die Methoden *Get()* und *Put()* ersetzt.

TC_812Controller
+ DWORD **oFlag** + DWORD **oIn** + DWORD **oOut1** + DWORD **oOut2**

+ DWORD **oDPRam**
+ **TC_812Controller** (DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList *Devices) + virtual BOOL **Check** () + char **Get** (DWORD addr) + void **Put** (DWORD addr, BYTE data) + int **PutChar** (const char c) + char **GetChar** (void) + int **ExecuteCmd** (LPTSTR pString)

(4) Klasse TC_832Controller

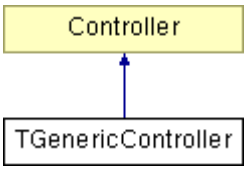


Die Klasse implementiert die Hardware/Treiberzugriffe für eine *C-832* Motorcontrollerkarte. Die globalen Funktionen *C832_Put()*, *C832_Get()*, *PutWord()*, *PutDWord()*, *GetWord()*, *GetDWord()*, *Drive628c()* und *LM628Ready()* werden durch entsprechende Methoden ersetzt.

TC_832Controller
+ BYTE **activeConfig** + WORD **activeDrive** + WORD **baddr** + BOOL **bLimitHit** + BOOL **bIdle** + BOOL **bIOActive** - const WORD **CmdRegister** - const WORD **DataRegister** - DWORD **dwSimBasePort**
+ **TC_832Controller** (DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList *Devices) + virtual BOOL **Check** () + void **UpdateController** (BYTE activeConfig, WORD activeDrive) + long **Drive628c** (BYTE cmd, WORD ctrl_word, long param, WORD base, WORD regaddr, unsigned short drive) + void **Put** (unsigned port, int value) + int **Get** (unsigned port) + BOOL **LM628Ready** (WORD base, unsigned short drive) + static void CALLBACK **LimitWatch** (UINT, UINT, DWORD, DWORD, DWORD) - int **GetWord** (WORD base, WORD regaddr, unsigned short drive)

- long **GetDWord** (WORD base, WORD regaddr, unsigned short drive) - void **PutWord** (int data, WORD base, WORD regaddr, unsigned short drive) - void **PutDWord** (long data, WORD base, WORD regaddr, unsigned short drive)

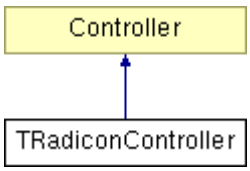
(5) Klasse TGenericController



Die Klasse implementiert die Hardware/Treiberzugriffe für die *AX5216* Zählerkarte. Sie übernimmt wesentliche Implementierungsdetails der Klasse *TAm9513* und ersetzt diese.

TGenericController
BYTE **nbChipId** - float **fTimeCorrection** - BOOL **bIsLatched** - BOOL **bIsStopped** - WORD **wLowTicks** - WORD **wHighTicks** - WORD **wLowCounts** - WORD **wHighCounts**
+ **TGenericController** (DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList *Devices) + virtual **~TGenericController** () + virtual BOOL **Check** () + int **Init** (float fTimeCorrection) + int **IOCTL** (TIOCCmd, DWORD &) + int **IOCTL** (TIOCCmd) + void **SetSound** (BOOL param) + DWORD **GetTicksPerSecond** (void) + void **ClearToggleOut** (BYTE) + void **SetToggleOut** (BYTE) - BOOL **SplitNumber** (DWORD, WORD &, WORD &)

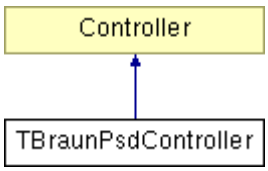
(6) Klasse TRadiconController



Die Klasse implementiert die Hardware/Treiberzugriffe für die *Radicon* Controllerkarte. Sie übernimmt wesentliche Implementierungsdetails der Klasse *TRadiconHW* und ersetzt diese.

TRadiconController
- DWORD **controlPort** - DWORD **dataPort**
+ **TRadiconController** (DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList *Devices) + virtual ~**TRadiconController** () + virtual BOOL **Check** () + int **UploadFirmware** (void) + int **SetParameters** (unsigned short upperThreshold, unsigned short lowerThreshold, int highvoltage, double exposureTime, unsigned long impulseCount, BOOL bSound) + int **Execute** (EOperationMode mode) + int **GetValues** (EReadoutMode, double *exposureTime, unsigned long *impulseCount) + void **reset** () - int **TransmitMessage** (int nFunctionCode, unsigned char *lpszMessage=0, int nMessageLength=0) - int **ReceiveMessage** (int nFunctionCode, unsigned char *lpszMessage, int nMaxMessageLength, int &nRealMessageLength) - int **TransmitFunctionCode** (int nFunctionCode) - BOOL **out_byte** (unsigned char d) - BOOL **in_byte** (unsigned char &d) - BOOL **IsReadyToRead** () - BOOL **IsReadyToWrite** () - DWORD **CurrentTime** (void)

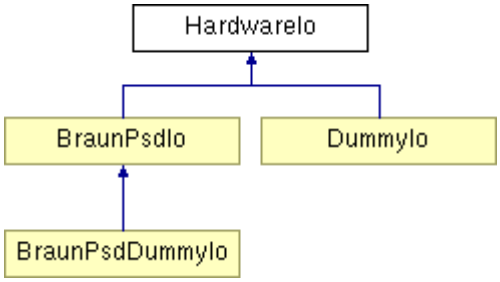
(7) Klasse TBraunPsdController



Die Klasse implementiert die Hardware/Treiberzugriffe für die *Braun-PSD* Controllerkarte.

TBraunPsdController
- BraunPsdIo * **SpecialHardware**
+ **TBraunPsdController** (DEVICETYPE DeviceID, LPTSTR HardwareID, DeviceList *Devices) + virtual **~TBraunPsdController** () + virtual BOOL **Check** () + BOOL **SetTimeout** (unsigned long Cycles) + BOOL **GetData** (IrpParamsGetData *Params)

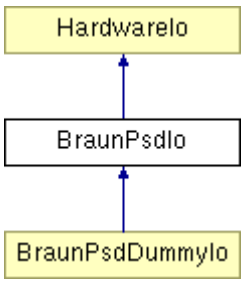
(8) Klasse HardwareIo



Die Klasse implementiert den eigentlichen Hardware/Treiberzugriff für die Controllerklassen.

HardwareIo
- char **szDeviceName** [257] - HANDLE **hDevice**
+ **HardwareIo** () + **HardwareIo** (char *szDeviceName) + virtual **~HardwareIo** () + virtual BOOL **Read** (DWORD Offset, BYTE *Data) + virtual BOOL **Read** (DWORD Offset, WORD *Data) + virtual BOOL **Read** (DWORD Offset, DWORD *Data) + virtual BOOL **Write** (DWORD Offset, BYTE Data) + virtual BOOL **Write** (DWORD Offset, WORD Data) + virtual BOOL **Write** (DWORD Offset, DWORD Data)

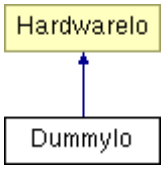
(9) Klasse BraunPsdIo



Diese Klasse erweitert *HardwareIo* um spezielle Funktionen, die von der Klasse *BraunPsdController* genutzt werden (erweiterte IOCTL-Schnittstelle des Treibers für *Braun-PSD*) (vergl. Spezifikation der XCTL-Gerätetreiber 2.1.2).

BraunPsdIo
<>
+ **BraunPsdIo** ()
+ **BraunPsdIo** (char *id)
+ virtual **~BraunPsdIo** ()
+ virtual BOOL **Read** (DWORD Offset, BYTE *Data)
+ virtual BOOL **Read** (DWORD Offset, WORD *Data)
+ virtual BOOL **Read** (DWORD Offset, DWORD *Data)
+ virtual BOOL **Write** (DWORD Offset, BYTE Data)
+ virtual BOOL **Write** (DWORD Offset, WORD Data)
+ virtual BOOL **Write** (DWORD Offset, DWORD Data)
+ virtual BOOL **SetTimeout** (unsigned long Cycles)
+ virtual BOOL **GetData** (IrpParamsGetData *Params)

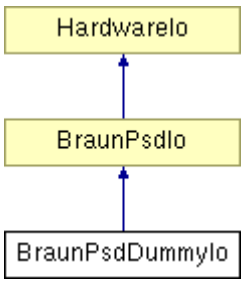
(10) Klasse DummyIo



Die Klasse dient als Standardhandler für Hardware/Treiberzugriffe für den Fall, dass ein entsprechender Treiber nicht korrekt geladen werden konnte. Dazu werden die Treiberzugriffe (Read/Write) abgefangen und als erfolgreich durchgeführt quittiert (*return(TRUE)*). Dieses Vorgehen ist zur Vermeidung von Systemfehlern (GPF) notwendig.

DummyIo
<>
+ **DummyIo** (int DeviceID, LPTSTR HardwareID) + virtual BOOL **Read** (DWORD Offset, BYTE *Data) + virtual BOOL **Read** (DWORD Offset, WORD *Data) + virtual BOOL **Read** (DWORD Offset, DWORD *Data) + virtual BOOL **Write** (DWORD Offset, BYTE Data) + virtual BOOL **Write** (DWORD Offset, WORD Data) + virtual BOOL **Write** (DWORD Offset, DWORD Data)

(11) Klasse BraunPsdDummyIo



Die Klasse ist eine Spezialisierung von *DummyIo* für die Klasse *BraunPsdController*. Die Notwendigkeit ihrer Implementierung ergibt sich ausschließlich als Konsequenz der Vererbungsbeziehung von *HardwareIo* und *BraunPsdIo*.

BraunPsdDummyIo
<>
+ **BraunPsdDummyIo** () + **BraunPsdDummyIo** (char *id) + virtual **~BraunPsdDummyIo** () + virtual BOOL **Read** (DWORD Offset, BYTE *Data) + virtual BOOL **Read** (DWORD Offset, WORD *Data) + virtual BOOL **Read** (DWORD Offset, DWORD *Data) + virtual BOOL **Write** (DWORD Offset, BYTE Data) + virtual BOOL **Write** (DWORD Offset, WORD Data) + virtual BOOL **Write** (DWORD Offset, DWORD Data) + virtual BOOL **SetTimeout** (unsigned long Cycles) + virtual BOOL **GetData** (IrpParamsGetData *Params)

(12) Klassen DeviceList und DeviceListEntry

Diese Klassen implementieren die Zuordnung der für *XCTL* konfigurierten Geräte (*HardwareID*) zu den im System installierten Treibern (*FileName*). Hierbei werden die ini-Parameter *IOAddr*, *BaseAddr* und *RamAddr* (je nach Art) zur eindeutigen Identifikation der Geräte genutzt.

DeviceListEntry
- DEVICETYPE **DeviceID** - char **HardwareID** [9] - char **FileName** [257] - DeviceListEntry * **prev** - DeviceListEntry * **next**
+ **DeviceListEntry** (DEVICETYPE DeviceID, LPTSTR HardwareID, LPTSTR FileName) + void **SetPrevEntry** (DeviceListEntry *pEntry) + void **SetNextEntry** (DeviceListEntry *pEntry) + DeviceListEntry * **GetPrevEntry** () + DeviceListEntry * **GetNextEntry** () + int **GetDeviceID** () + LPTSTR **GetHardwareID** () + LPTSTR **GetFileName** ()

DeviceList
- DeviceListEntry * **first** - DeviceListEntry * **last**
+ **DeviceList** () + **~DeviceList** () + void **Add** (DEVICETYPE DeviceID, LPTSTR HardwareID, LPTSTR FileName) + void **Add** (DeviceListEntry *pEntry) + void **Clear** () + LPTSTR **GetFileName** (int DeviceID, LPTSTR HardwareID)

Zusätzlich zu den bisher genannten Klassen wurden Funktionen zur Verwaltung der Gerätetreiber in XCTL implementiert.

Funktionen zur Treiberverwaltung		
Funktionsname	**Modul**	**Funktion**
EnumDevices	Hardwarezugriff	liefert die Anzahl der installierten Treiber einer Geräteklasse
GetDeviceDriver	Hardwarezugriff	liefert den (Zugriffs-)Namen eines bestimmten Treibers
RegisterDrivers	Hardwarezugriff	erstellt eine Tabelle aus Treibername,

		zugehöriger XCTL-interner Geräteklasse und HardwareID (DeviceList)
GetController	Motorsteuerung	liefert das zu einer HardwareID gehörige Controller-Objekt (Motoren)
GetController	Detektornutzung	liefert das zu einer HardwareID gehörige Controller-Objekt (Detektoren)

Neben diesen Neuimplementierungen sind Anpassungen des bestehenden Quelltextes notwendig.

Damit die Gerätetreiber von *XCTL* genutzt werden können, müssen die Zugriffsbezeichner der entsprechenden Treiber ermittelt und auf die XCTL-Geräteklassen (*RADICON*, *GENERIC*, *BRAUN*, *C812*, *C832*) abgebildet werden. Diese Aufgabe übernimmt die Funktion *RegisterDrivers()*, die dafür in den *DLL_PROCESS_ATTACH*-Zweigen der *DllMain()*-Funktionen von *Counters.dll* und *Motors.dll* aufgerufen werden muss.

Die eigentliche „Umstellung“ der direkten Hardwarezugriffe auf Treiberzugriffe erfolgt in den Klassen *TRadicon*, *TGenericDetector*, *TBraunPsd*, *TC812ISA* und *TC832* (Klassendiagramme in Anhang E).

(1) Einführen eines private-Members *Hardware* vom Typ *x*Controller (*x* steht für den jeweiligen Klassennamen) in den o.g. Klassen.

(2) Aufruf der Funktion *GetController()* in den Konstruktoren der o.g. Klassen. Wichtigster Parameter ist dabei die *HardwareID* des durch die Klasse repräsentierten Gerätes, da hierdurch die Zuordnung des entsprechenden Treibers erfolgt.

(3) Umstellung aller direkten Hardwarezugriffe auf die Methoden *Read()* und *Write()* des Members *Hardware* in den o.g. Klassen. Zu beachten ist hierbei, dass nicht mehr über absolute Adressen kommuniziert wird, sondern über Offsets der für den Treiber konfigurierten Basisadresse.

Damit sind die grundlegenden Anpassungen in XCTL abgeschlossen.

Beschreibung der Umsetzung von Hardwarezugriffen im Portierungsergebnis

Der Vorgang *Hardwarezugriff* gliedert sich in zwei Phasen (siehe Abbildung [5.3]).

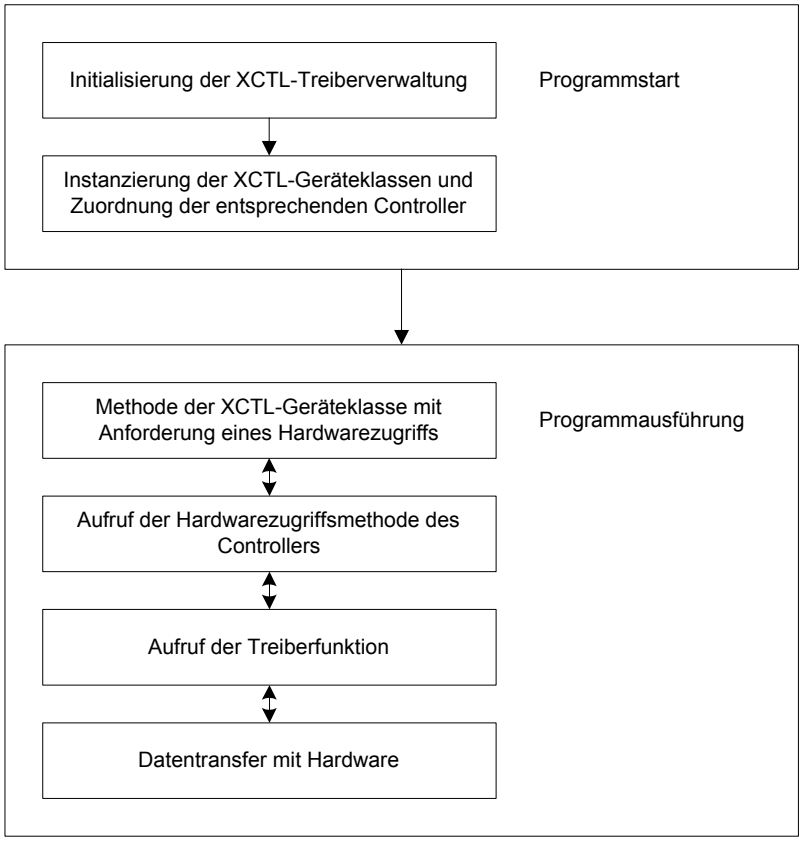


Abbildung [5.3] Phasen des Hardwarezugriffs von XCTL

Damit während der Programmausführung Hardwarezugriffe durchgeführt werden können, müssen während des Programmstarts einige notwendige Vorbereitungen getroffen werden. (siehe Abb. [5.4])

(1) Initialisierung der XCTL-Treiberverwaltung

Zugriffe auf Treiber werden unter Windows über die Systemfunktion *DeviceIoControl()* realisiert. Die Funktion benötigt als Parameter, neben dem Funktionscode der im Treiber auszuführenden Funktion, ein Handle auf den entsprechenden Treiber, welches durch die Systemfunktion *CreateFile()* erzeugt wird. Diese Funktion wiederum benötigt den Namen des Treibers. Da dieser den Nutzern des Treibers i.A. nicht bekannt ist und sich auch von Instanz zu Instanz unterscheidet, bietet das System die

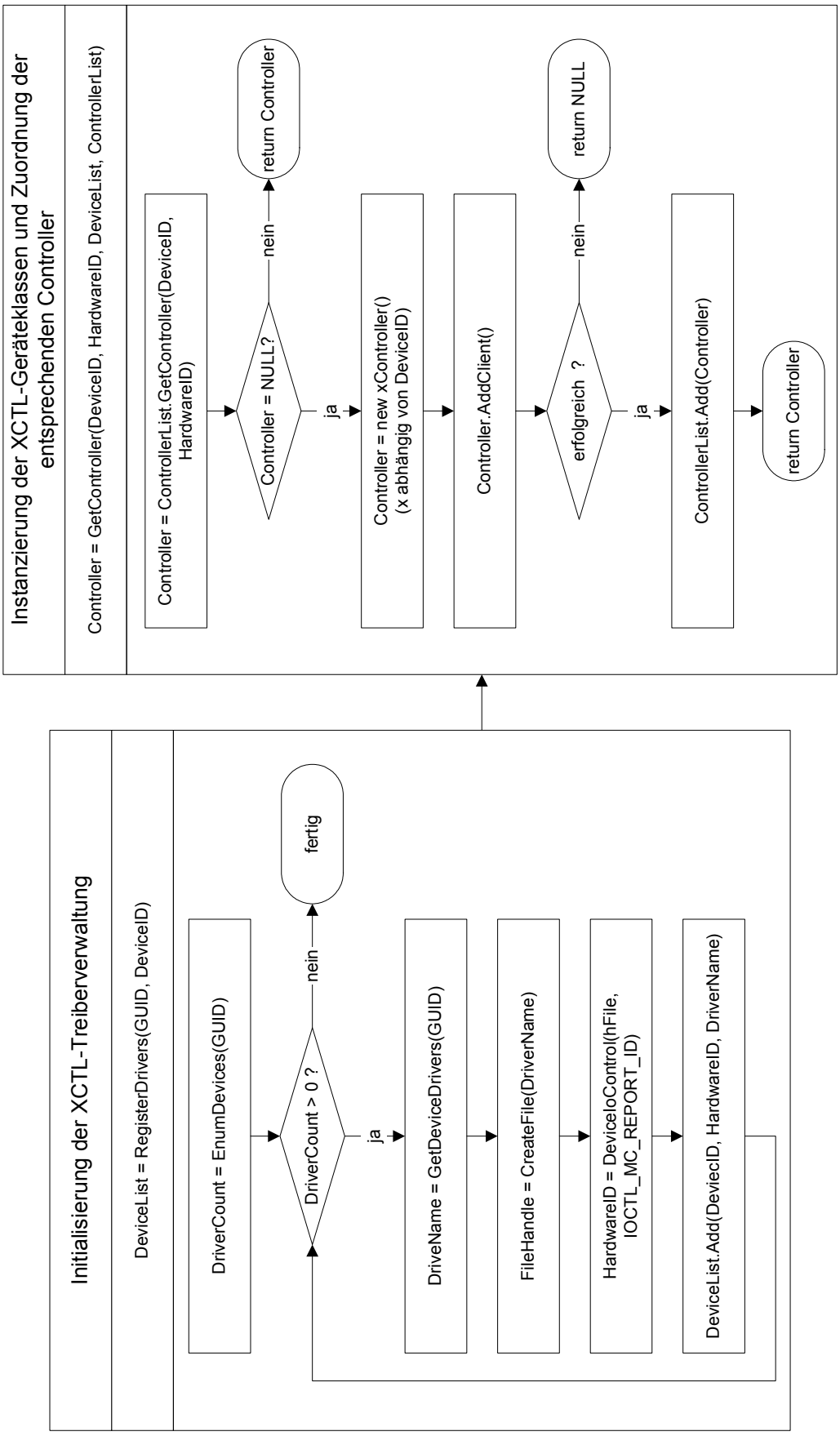
Möglichkeit, Treiber über einen allgemeineren Mechanismus zu identifizieren. Jeder Treiber besitzt unabhängig von seinen Instanzen eine s.g. *GUID* über die er sicher erkannt werden kann. Ist dem Nutzer des Treibers die entsprechende GUID bekannt, so kann er über Systemfunktionen (*EnumDevices()* und *GetDeviceDrivers()*) die Namen der gestarteten Treiberinstanzen ermitteln.
Diesen Mechanismus nutzt XCTL in der Startphase, um die benötigten Treiberinstanzen zu identifizieren und den XCTL-Geräteklassen zuzuordnen. Die einzelnen Instanzen werden über ihre konfigurierten Bezeichner (*HardwareID* = eingestellte Basisadresse des Treibers) unterschieden. Die Zugriffsdaten der Treiberinstanzen werden in der Datenstruktur *DeviceList* gespeichert, wobei die Informationen für Detektortreiber und Motortreiber in verschiedenen Listen verwaltet werden.

(2) Instanzierung der XCTL-Geräteklassen und Zuordnung der entsprechenden Controller

Die Controllerklassen (*TC812Controller*, ...) abstrahieren für die XCTL-Geräteklassen (*TC_812ISA*, ...) von den Hardwaresteuerungskarten. Jedem Objekt einer Geräteklasse wird eine entsprechende Instanz einer Controllerklasse zugeordnet, da über diese Hardwarezugriffe realisiert werden. Die erstellten Controllerobjekte werden in der Datenstruktur *ControllerList* verwaltet (getrennt für Detektor- und Motorcontroller).
Jede Steuerungskarte kann eine spezifische Anzahl von angeschlossenen Geräten kontrollieren. Dementsprechend kann einem einzelnen Controllerobjekt auch nur diese spezifische Anzahl von Geräteobjekten zugeordnet werden.

Den Ablauf der Hardwarezugriffe zur Laufzeit von XCTL verdeutlicht Abbildung [5.5]

Abbildung [5.4](folgende Seite)


Initialisierung der XCTL-Treiberverwaltung
DeviceList = RegisterDrivers(GUID, DeviceID)
DriverCount = EnumDevices(GUID)
DriverCount > 0 ?
nein
fertig
ja
DriveName = GetDeviceDrivers(GUID)
FileHandle = CreateFile(DriverName)
HardwareID = DeviceIoControl(hFile, IOCTL_MC_REPORT_ID)
DeviceList.Add(DeviecID, HardwareID, DriverName)
Instanzierung der XCTL-Geräteklassen und Zuordnung der entsprechenden Controller
Controller = GetController(DeviceID, HardwareID, DeviceList, ControllerList)
Controller = ControllerList.GetController(DeviceID, HardwareID)
Controller = NULL?
nein
return Controller
ja
Controller = new xController() (x abhängig von DeviceID)
Controller.AddClient()
erfolgreich ?
nein
return NULL
ja
ControllerList.Add(Controller)
return Controller

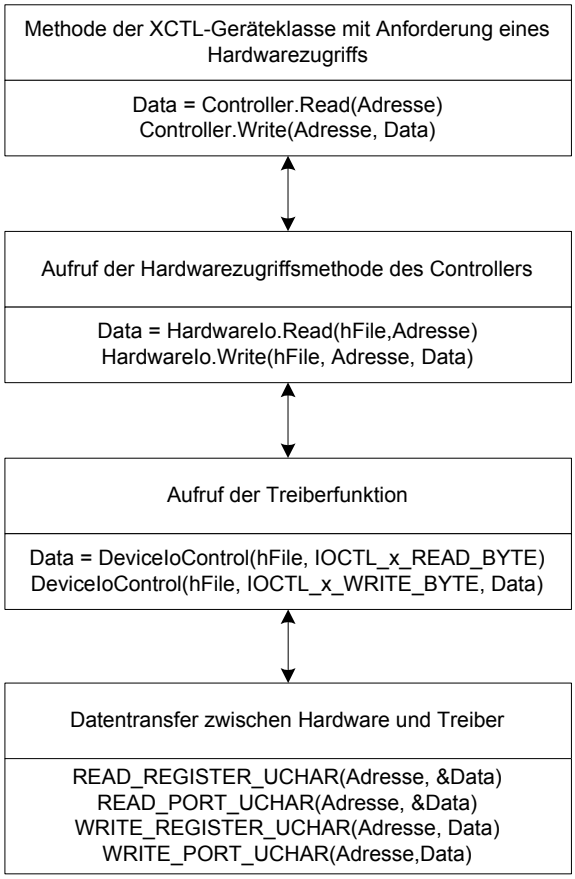


Abbildung [5.5]

6. Test

Testziel

Der Erfolg einer Portierung wird erst durch einen erfolgreichen Test des Portierungsgegenstandes auf der Zielplattform sichtbar und dokumentiert. Nachzuweisen ist hierbei einerseits die uneingeschränkte Erhaltung der Funktionalität des Hostsystems durch das Zielsystem und andererseits die Einhaltung bestimmter Qualitätsanforderungen bezüglich der Existenz von Fehlern im portierten Softwaresystem.

Möglichkeiten der Testdurchführung

Grundlage einer effizienten Testdurchführung ist der Entwurf einer adäquaten, auf die Zielstellung des Tests fokussierten Teststrategie und somit die Auswahl eines geeigneten Testverfahrens. Da eine Vielzahl allgemein anerkannter Verfahren existiert, wird im Folgenden eine kurze Einführung in die grundlegenden Testverfahren gegeben und anschließend die Entscheidung für das letztendlich eingesetzte Verfahren erläutert.

Software-Prüftechniken lassen sich prinzipbedingt in zwei Oberklassen einteilen: *statische* Testverfahren und *dynamische* Testverfahren [Liggesmeyer02].

Statische Testverfahren

Statische Verfahren beziehen sich ausschließlich auf den Quelltext des zu untersuchenden Softwaresystems und werden weiter in *Analyseverfahren* und *formale Verfahren* unterteilt.

Analyseverfahren zielen darauf ab, Aussagen über Produkteigenschaften zu treffen, problematische Konstrukte im Quelltext zu identifizieren, Quelltexteigenschaften anzugeben und somit den Entwicklungsprozess effizient zu steuern. Aussagen über Korrektheit oder Zuverlässigkeit der Software können hierbei nicht getroffen werden.
Dem entgegen liefern formale Verfahren vollständige Aussagen zur Korrektheit eines Programms mit Hilfe formaler Mittel. Dabei soll die

Konsistenz zwischen der Software und deren Spezifikation bewiesen werden.

Dynamische Testverfahren

Bei dynamischen Verfahren wird das zu testende Softwaresystem ausgeführt, mit Eingabedaten versehen und die erzeugten Ausgabedaten werden analysiert. Grundgedanke dynamischer Tests ist die Annahme, dass im Quelltext statisch vorhandene Fehler ein erkennbares Fehlverhalten während der Programmausführung bewirken. Basis dieser Testverfahren ist also die Erzeugung repräsentativer, fehlersensitiver und redundanzarmer Testfälle. Allen dynamischen Verfahren ist gemein, dass es sich um Stichprobenverfahren handelt und diese somit die Korrektheit der getesteten Software nicht beweisen können.

Die drei wichtigsten Klassen dynamischer Prüftechniken sind *strukturorientierte* Tests, *funktionsorientierte* Tests und *diversifizierende* Tests.

(1) Strukturorientierte Tests

Die Klasse der strukturorientierten Prüftechniken ist ihrerseits noch einmal in *kontrollflussorientierte* und *datenflussorientierte* Verfahren unterteilt. Wesentlicher (und namensgebender) Unterschied ist die Fokussierung auf Prüfung der programminternen Kontrollstrukturen (kontrollflussorientiert) bzw. Prüfung der programminternen Strukturen für Datenzugriffe und -manipulation (datenflussorientiert). Korrektheitskriterium ist die Konformität der Programmausgabedaten bezüglich der Spezifikation des zu testenden Softwaresystems. Die Testvollständigkeit wird jeweils durch den Grad der Abdeckung der entsprechenden Quelltextelemente durch das Ausführen der spezifizierten Testfälle bewertet. Strukturorientierte Testtechniken definieren keine Regeln für die Testfallerzeugung.

(2) Funktionsorientierte Tests

Bei funktionsorientierten Testtechniken stehen die in der Spezifikation des Softwaresystems definierten Funktionen im Mittelpunkt des Testvorgangs. Solche Tests werden auch als Black-Box-Tests bezeichnet,

da die innere Struktur der Software bei der Erstellung der Testfälle keinerlei Berücksichtigung erfährt und ausschließlich das von außen beobachtbare Verhalten den Testentwurf bestimmt. Anhand der Softwarespezifikation werden nach bestimmten Regeln Testfälle und Testdaten generiert, mit denen die Software ausgeführt wird. Die Reaktionen der Software auf diese Eingabedaten werden dann entsprechend der Spezifikation bewertet.

(3) Diversifizierende Tests

Im Gegensatz zu den vorher genannten Verfahren wird ein Softwaresystem hierbei nicht gegen seine Spezifikation getestet, sondern gegen andere Versionen der Software, in dem die Testergebnisse verschiedener Softwareversionen miteinander verglichen werden. Vorteil dieser Vorgehensweise ist die Vermeidung der aufwändigen und teilweise nur schwer realisierbaren Bewertung der Korrektheit der Testergebnisse bezüglich der Spezifikation. Auch kann der Ergebnisvergleich leicht automatisiert werden, was zu einer weiteren Effizienzsteigerung des Testvorgangs führt.

In der Literatur werden im Wesentlichen drei diversifizierende Testverfahren unterschieden:

(3.1) Back-to-Back-Test

Diesem Test liegt das Prinzip der heterogenen Redundanz zu Grunde. Dem entsprechend werden mindestens zwei Versionen des Softwaresystems miteinander verglichen, die auf der gleichen Spezifikation basieren, aber von unterschiedlichen und unabhängigen Entwicklerteams implementiert wurden. Da diese Versionen ggf. eine völlig verschiedene interne Struktur besitzen, besteht eine gewisse Wahrscheinlichkeit, dass sie nicht die gleichen systematischen Fehler enthalten und somit durch Vergleich der Testergebnisse ein Fehlverhalten des Softwaresystems erkannt werden kann. Fehler, die in allen Versionen der Software enthalten sind, können so entsprechend nicht gefunden werden. Auch ist bei einer Divergenz der Testergebnisse nicht klar, welche Version spezifikationskonform ist und welche den Fehler aufweist.

(3.2) Mutationen-Test

Beim diesem Testverfahren werden die zu vergleichenden Versionen aus einer Basisversion, die als fehlerfrei angesehen wird, durch gezieltes Einfügen eines genau bekannten Fehlers erzeugt. Diese Versionen werden als Mutationen bezeichnet. Mit Hilfe dieser Mutationen können dann Testtechniken auf ihre Wirksamkeit hin untersucht werden.

(3.3) Regressionstest

Ziel dieses Tests ist es, nachzuweisen, dass die an einem Softwaresystem durchgeführten Modifikationen keine unerwünschten Auswirkungen auf die Funktionalität des Systems besitzen. Als Modifikation wird hierbei z.B. das Hinzufügen neuer Funktionalität, das Beseitigen von Fehlern oder auch die Anpassung an veränderte Betriebsumgebungen verstanden. Durch diese Eingriffe in die Software können durchaus neue Fehler in zuvor fehlerfreie Teile der Software hineingetragen werden.
Theoretisch müssten nach einer Modifikation sämtliche spezifizierten Testfälle auf die neue Version angewendet werden. In der Praxis kann jedoch oft sichergestellt werden, dass schon das Ausführen einer Teilmenge der Testfälle für einen zuverlässigen Test ausreicht. Ob ein Testfall korrekte Ergebnisse liefert, wird auch hier nicht durch einen Vergleich mit der Spezifikation bestimmt, sondern durch den Vergleich der spezifischen Testergebnisse mit denen einer Vorgängerversion. Wichtige Vorraussetzung für einen zuverlässigen Regressionstest ist die Einhaltung jeweils gleicher Ausgangsbedingungen (Eingabedaten, etc.) bei der Durchführung der Testvorgänge.

Wahl des Testverfahrens zur Überprüfung des Portierungserfolges

Die Wahl des einzusetzenden Testverfahrens wird nun im Wesentlichen durch die oben definierten Testziele und Effizienzüberlegungen bestimmt.

Prinzipiell sind die genannten dynamischen Testverfahren und die formellen statischen Verfahren mehr oder weniger geeignet, die definierten Testziele zu überprüfen, wobei der mit dem jeweiligen Testverfahren verbundene Aufwand stark variiert. Aus diesem Grund fließen nun Effizienzüberlegungen in das Auswahlverfahren ein.

Da der Portierungsgegenstand auf dem Hostsystem keineswegs den Anspruch erhebt, fehlerfrei zu sein, wäre es ineffizient, dies von der portierten Version zu fordern. Somit ist ein Testverfahren zu wählen, welches speziell jenes Fehlverhalten aufzeigt, dass direkt auf den Portierungsvorgang zurückzuführen ist. Hierbei ist anzumerken, dass dies auch Fehlverhalten umfasst, welches erst durch die Portierung zu Tage tritt, dessen ursächlicher Fehler aber bereits in der Portierungsbasis enthalten ist. Dieser Anforderung entsprechen die diversifizierenden Testverfahren am besten und unter diesen Verfahren speziell der Regressionstest.

Testverfahren

Mit Hilfe eines umfangreichen Tests soll nun gezeigt werden, dass die Funktionalität des Portierungsgegenstandes durch die Portierung nicht verändert wurde und das Portierungsergebnis die gleichen Qualitätskriterien erfüllt wie die Portierungsbasis.
Der Test wird aufgrund der Komplexität des XCTL-Systems und der Art und Weise, wie die Portierung durchgeführt wurde, in zwei Bereiche aufgeteilt. In Abbildung [6.1] ist diese Aufteilung dargestellt.

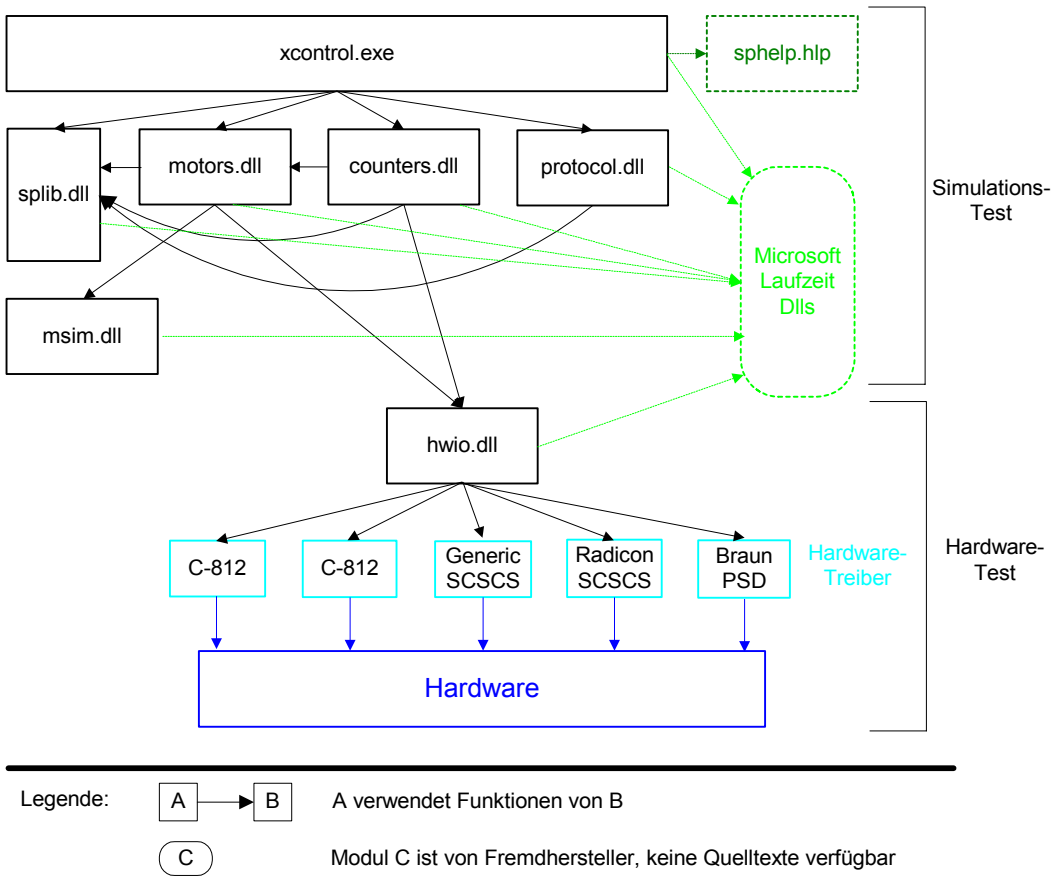


Abbildung [6.1] Partitionierung des Tests

Für den Test werden zum Teil die schon als Regressionstest anerkannten Testsequenzen[1] des ATOS-Testsystems[2] genutzt, siehe Tabelle [6.1]. Für diesen Teil des Tests lässt sich der Portierungsgegenstand in einem Simulationsmodus ausführen, in dem sämtliche Funktionalität prinzipiell zur Verfügung steht. Dabei werden Hardwarezugriffe allerdings von speziellen Softwarefunktionen/Klassen abgefangen, welche die entsprechende Hardware emulieren.
Ein Ausführen der ATOS-Testsequenzen in diesem Modus wurde bisher als Regressionstest der Funktionalität des Portierungsgegenstandes akzeptiert, so dass davon ausgegangen werden kann, dass dies auch für die portierte Version gilt.
Da ein wesentlicher Teil des Portierungsaufwandes durch die Adaption der Hardwaresteuerung verursacht wurde und dies auch den größten Teil der Quelltextmodifikationen ausmacht, kann ein Regressionstest in der simulierten Hardwareumgebung nicht als ausreichend angesehen werden.

[1] Herkunft aus dem Projekt CVS
[2] ATOS ist ein Werkzeug zum automatisierten Testen von Programmen, siehe [ATOS 02], Anhang C - Werkzeuge, ATOS

Es ist also zwingend erforderlich, spezialisierte Tests in einer realen Hardwareumgebung durchzuführen. Hierfür werden aus den ATOS-Testsequenzen des Regressionstests dedizierte Testsequenzen für den Test der Hardwaresteuerung unter Realbedingungen abgeleitet.

Ein Teil der Funktionalität des Portierungsgegenstandes wurde im Zuge der Portierung in externe Softwaremodule (Gerätetreiber) ausgelagert. Diese Module sind ein Teil des portierten Softwaresystems und wesentlich für die Gesamtfunktionalität. Daher müssen diese speziell getestet werden.
Für neu erstellte, eigenständige Softwaremodule macht ein Regressionstest keinen Sinn. Aus diesem Grund werden die Treiber funktionsorientierten Black-Box-Tests unterzogen. Das Testen der Gerätetreiber geschieht in zwei Schritten. Zum einen werden mit einer entsprechenden Testsoftware die allgemeinen Verwaltungsfunktionen und das Verhalten im System getestet. Ein entsprechendes Testprogramm wird von Microsoft zur Verfügung gestellt. Zum anderen wird die IOCTL-Schnittstelle und entsprechend die hardwarebezogene Funktionalität durch ein selbst entwickeltes Testprogramm geprüft.
Eine Besonderheit bezüglich des Tests der IOCTL-Schnittstelle stellt der Gerätetreiber für den Braun-PSD dar. Dieser Treiber besitzt eine funktionell erweiterte Schnittstelle, die entsprechend speziell getestet werden muss (siehe Kapitel 5 – Spezifikation der Treiber). Da die hinter den zusätzlichen Befehlscodes *IOCTL_DC_SET_WFR_CYCLES* und *IOCTL_DC_GET_DATA* stehende Funktionalität nur in einem spezifischen Kontext getestet werden kann, ist es effizienter, diese in der portierten Version von XCTL zu testen.

In der folgenden Abbildung [6.2] ist das Testvorgehen zusammenfassend dargestellt.

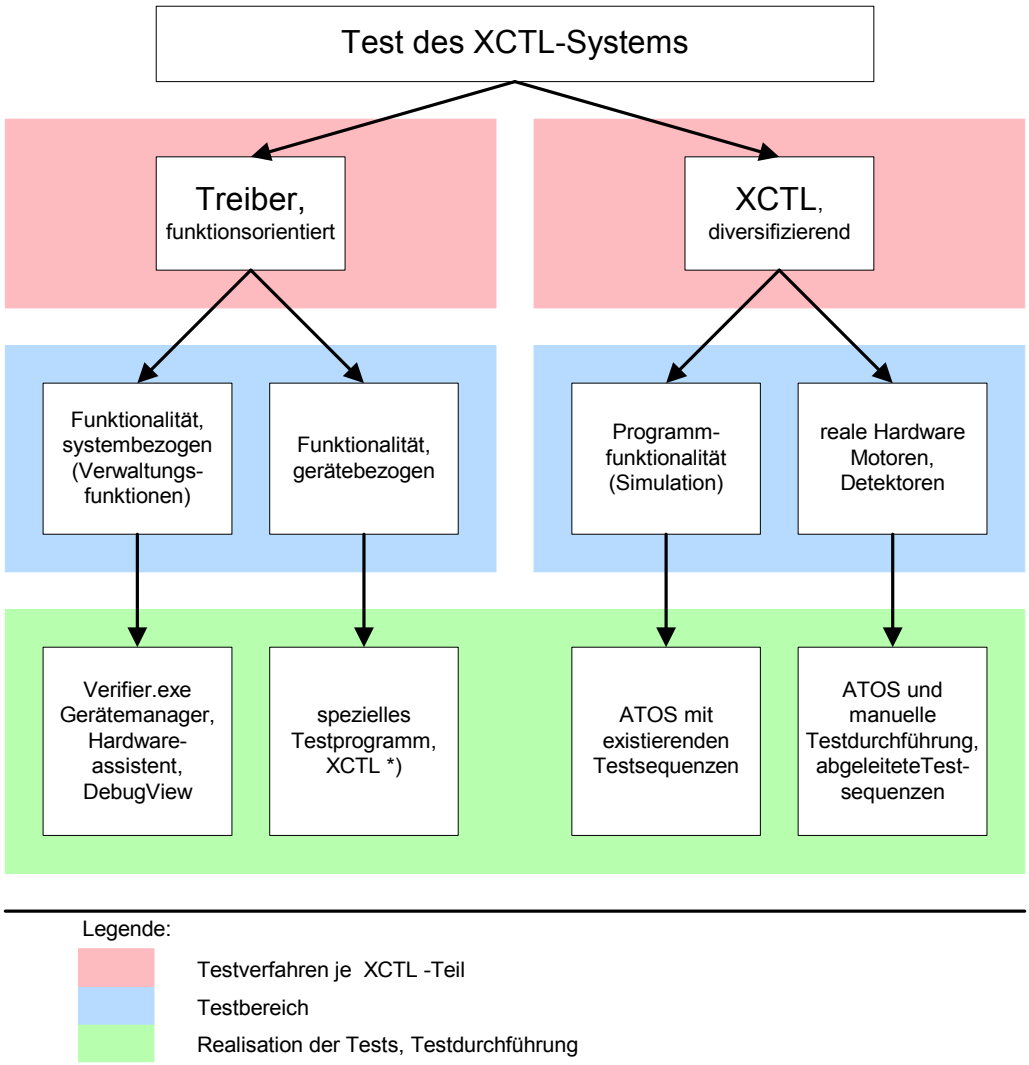


Abbildung [6.2] Testvorgehen

Testvorbereitung

Analyse der bisherigen ATOS-Testsequenzen und Ableiten der dedizierten Hardware-Testfälle

Ausgangspunkt des Tests sind die für *XCTL* definierten ATOS-Testsequenzen, auf deren Grundlage bisher die Regressionstests durchgeführt wurden. Testgegenstand dieser Sequenzen ist *XCTL* in einer simulierten Hardwareumgebung.

Zur Erstellung der hardwarebezogenen Testfälle ist eine Untersuchung der bisher für *ATOS* definierten Testsequenzen auf Nutzung der Motor- bzw. Detektorhardware notwendig. Alle Teilschritte von Testsequenzen, die keine entsprechenden Hardwarezugriffe verursachen, werden lediglich in der Simulation getestet. Für Teilschritte mit Hardwarenutzung werden neue Testsequenzen für den Test unter Realbedingungen entworfen. In Tabelle [D.5] ist die entsprechende Auswertung der Testsequenzen dargestellt. Die daraus abgeleiteten Testfälle sind in Tabelle [6.1] aufgeführt und in Anhang G in Form von ATOS-Testsequenzen definiert.

abgeleitete Testsequenzen			
Anwendungsfall	**Testfall**	**Hardware**	**Name der Testsequenz**
Automatische Justage	Ausführen des Vorgangs „Automatische Justage“	C-812, C-832, Radicon SCSCS	Test_AJ.H.1
Area-Scan	Ausführen des Vorgangs „Area-Scan“	C-812, C-832, Braun PSD	Test_ARS.H.1
Detektornutzung (0-dimensionale)	Ausführen des Dialogs „Zähler-Konfiguration“ und Anzeige der Messwerte	Radicon SCSCS	Test_D0.H.1
Detektornutzung (0-dimensionale)	Ausführen des Dialogs „Zähler-Konfiguration“ und Anzeige der Messwerte	Generic SCSCS	Test_D0.H.2
Detektornutzung (1-dimensionale)	Ausführen des Dialogs „Zähler-Konfiguration“ und Anzeige der Messwerte	Braun PSD	Test_D1.H.1
Halbwertsbreite messen	Ausführen des Vorgangs „Halbwertsbreite messen“	C-812, C-832, Radicon PSD	Test_HWB.H.1
Line-Scan	Ausführen des Vorgangs „Line-Scan“	C-812, C-832, Radicon SCSCS	Test_LS.H.1
Manelle Justage	Positionieren eines Motors durch Direktbetrieb (neuer Winkel, relative Null setzen, relative Null aufheben)	C-812	Test_MJ.H.1.1
Manuelle Justage	Positionieren eines Motors durch Direktbetrieb (neuer Winkel, relative Null setzen, relative Null aufheben)	C-832	Test_MJ.H.1.2
Manuelle Justage	Positionieren eines Motors mittels Cursor-Tasten und Scrollbar (Schritt-Betrieb)	C-812	Test_MJ.H.2.1

Manuelle Justage	Positionieren eines Motors mittels Cursor-Tasten und Scrollbar (Schritt-Betrieb)	C-832	Test_MJ.H.2.2
Manuelle Justage	Positionieren eines Motors mittels Cursor-Tasten und Scrollbar (Fahr-Betrieb)	C-812	Test_MJ.H.3.1
Manuelle Justage	Positionieren des Motors mittels Cursor-Tasten und Scrollbar (Fahr -Betrieb)	C-832	Test_MJ.H.3.2
Motorsteuerung	Ausführen des Vorgangs „Direkte Steuerung...“ für Motor	C-812	Test_MS.H.1.1
Motorsteuerung	Ausführen des Vorgangs „Direkte Steuerung...“ für Motor	C-832	Test_MS.H.1.2
Motorsteuerung	Ausführen des Vorgangs „Referenzpunktlauf“ für Motor	C-812	Test_MS.H.2.1
Motorsteuerung	Ausführen des Vorgangs „Optimieren...“ für Motor	C-812	Test_MS.H.3.1
Motorsteuerung	Ausführen des Vorgangs „Optimieren...“ für Motor	C-832	Test_MS.H.3.2

Tabelle [6.1]

Bemerkungen zu den abgeleiteten Testfällen:

(1)
Sämtliche Testfälle für den Detektor *Radicon SCSCS* sind prinzipiell auch für den Detektor *Generic SCSCS* möglich, da es sich ebenfalls um einen 0-dimensionalen Detektor handelt. Allerdings ist die Implementierung bzw. physische Anbindung des Detektors nicht fehlerfrei, was sich darin zeigt, dass die mit XCTL ermittelten Messwerte nicht mit denen des externen Strahlenmessgerätes *RFT 20046* korrespondieren. Der Detektor ist also für quantitative Messungen nicht wirklich einsatzfähig, was auch von den Nutzern[3] bestätigt wurde. Seine grundlegende Funktionalität in XCTL wird mit dem Anwendungsfall *Detektornutzung* (Test_D0.H.2) geprüft.

(2)
Die Ansteuerung des Detektors *Braun-PSD* ist zur Zeit fehlerhaft implementiert, so dass keine gültigen Messwerte mit XCTL ermittelt werden können. Der Anwendungsfall *Halbwertsbreite messen* kann aus diesem Grund für diesen Detektor nicht durchgeführt werden. Auch der Anwendungsfall *Area-Scan* ist entsprechend nur bedingt durchführbar.

[3] Technische Mitarbeiterin Jane Richter

(3)
Die prinzipielle Funktionalität der Motorsteuerung wird in der Simulation getestet, so dass für den Test der Hardwareansteuerung spezialisierte synthetische Tests ausreichen, die mit den Testmotoren (C-812-Schnittstelle, C-832-Schnittstelle) durchgeführt werden. Die korrekte Adressierung der Motoren wird in einem speziellen Test überprüft.

(4)
Der Testfall *Referenzpunktlauf* kann für die Controllerkarte *C-832* nicht durchgeführt werden, da sämtliche den Autoren vorliegende Konfigurationsdateien der Realumgebung diesen Anwendungsfall explizit deaktivieren und der entsprechende Testmotor nicht über die notwendigen Endlagenschalter verfügt.

(5)
Der Anwendungsfall *Topographie* wird nur in der Simulation getestet, da er sich nicht wesentlich vom Anwendungsfall *Line-Scan* unterscheidet und in der Realumgebung Referenzdaten nur unter großem Aufwand gewonnen werden können (Langzeitbelichtung einer Fotoplatte).

Generierung von Referenzdaten

Ein Teil der hardwarebezogenen Testfälle benötigt zur Validierung des Testergebnisses entsprechende Referenzdaten, die durch das Ausgangssystem erzeugt wurden.

Da das verwendete Testwerkzeug *ATOS* selbst eine 32-bit Anwendung ist, kann ein durch *ATOS* unterstützter Test nur unter einem Win32 System durchgeführt werden. Die Abwärtskompatibilität von Windows 2000 gestattet es zwar, auch 16-bit Anwendungen auszuführen, allerdings mit der Einschränkung, dass keine direkten Hardwarezugriffe stattfinden können. Zur Gewinnung von realen Referenzdaten ist also ein Ausführen von *XCTL* und manuelles Abarbeiten der Testsequenzen unter Windows 3.11 notwendig.

Anpassung des Testsystems

Für den reibungslosen Ablauf des Testvorgangs sind kleinere Anpassungen des Testsystems notwendig. So benötigt *ATOS* für die Steuerung der Oberflächenelemente des Testgegenstandes Zugriff auf die entsprechenden Ressourceninformationen. Da für das Portierungsergebnis solche Informationen noch nicht in ATOS-kompatibler Form (*URF*) vorliegen, müssen diese mit dem *RC-Parser* generiert und in *ATOS* eingebunden werden. Trotz dieser Maßnahme werden die Standarddialoge (z.B. *Datei öffnen...*, *Datei speichern...*) von Windows 2000 durch ATOS nicht korrekt erkannt, so dass die Testsequenzen umgeschrieben werden müssen. An Stelle der automatisierten Eingabe durch ATOS tritt hier eine manuelle Eingabe durch den Tester.

Ein weiteres Problem ergibt sich aus der Art der Testdurchführung selbst. ATOS arbeitet die Testsequenzen nicht ereignis- sondern zeitgesteuert ab, was dazu führt, dass die spezifische Ausführungsgeschwindigkeit der Testplattform zu einem testentscheidenden Faktor wird. Sollte die Ausführung eines Testfalls plattformbedingt einen längeren Zeitraum benötigen als vom Autor der entsprechenden Testsequenz vorgesehen[4], so schlägt dieser Testfall unabhängig von seinem eigentlichen Ergebnis fehl, was sowohl Effektivität und Effizienz des Testvorgangs negativ beeinflusst. Um dies zu verhindern, ist ein Abschätzen der auf der konkreten Testplattform für die Ausführung eines Testfalls benötigten Zeit und die entsprechende Anpassung der Testsequenzen notwendig. Hilfreich hierbei sind die bei manueller Abarbeitung der Testsequenzen in der Portierungsbasis gewonnenen Zeitinformationen.

Für die Tests in der Realumgebung ist es außerdem notwendig, spezielle an die Rahmenbedingungen der Messplätze angepasste Konfigurationsdateien für den Testgegenstand zu erstellen.

[4] ...weil die Ausführung des Testfalls auf der Referenzplattform eben zufällig diese Zeit benötigt hat...

Testdurchführung

Dem definierten Testverfahren entsprechend gliedert sich der Testvorgang in verschiedene Phasen. Basis der späteren Testabschnitte ist die Überprüfung der fundamentalen Funktionalität der Hardwaretreiber. Hierauf aufbauend folgen der Test der grundlegenden Funktionalität von *XCTL* in einer simulierten Umgebung und der Test der hardwareabhängigen Anwendungsfunktionalität in einer Realumgebung.

Test der Hardwaretreiber

Das Testen der Verwaltungsfunktionen eines Treibers und des Verhaltens im System stellt eine besondere Aufgabe dar. Durch die Implementation der Verwaltungsfunktionen wird ein Treiber mit einer Schnittstelle ausgerüstet, über die Betriebssystem und Treiber kommunizieren können. Diese Schnittstelle unterliegt einer von Microsoft vorgeschriebenen Spezifikation [MSDN 03b], gegen die aufgrund der tiefen Verflechtung von Betriebsystem und Treibern nur schwer getestet werden kann. Speziell kommt hier die Tatsache zum Tragen, dass es sich bei einem Treiber nicht um ein eigenständig ausführbares Programm handelt, sondern vielmehr um einen Teil des Betriebssystems, der nur in diesem bestimmten Kontext seine Funktionalität zur Verfügung stellt. Glücklicherweise hat auch Microsoft dies erkannt und jeder Windows 2000 Distribution ein Testprogramm (siehe Anhang C - Verifier.exe) beigelegt, mit dessen Hilfe beliebige Treiber gegen die Schnittstellenspezifikation in Form von Implementierungsrichtlinien[5] getestet werden können. Auch das Verhalten im System, wie z.B. das Anfordern und Freigeben von Systemspeicher oder das Überschreiten von Speichergrenzen, kann mit diesem Werkzeug zuverlässig und einfach untersucht werden. Da die Umsetzung der Tests mit dem Werkzeug kein Wissen über den inneren Aufbau eines Treibers erfordert, handelt es sich um funktionsorientierte Black-Box-Tests.

[5] vergl. [MSDN 03b], [Solomon 00], [Oney 03]

Testablauf:

(1) Im Testwerkzeug *Verifier* wird unter "Einstellungen" der (zuvor installierte) zu testende Treiber für die Überprüfung ausgewählt.
(2) Der Überprüfungstyp wird ausgewählt
(3) Abschließend müssen die Änderungen übernommen, das Programm beendet und Windows neu gestartet werden.

Fortan wird der zu testende Treiber vom Systemstart bis zum Herunterfahren des Systems überwacht. Die Überwachung kann mit dem Testwerkzeug wieder deaktiviert werden. Wenn durch die Testumgebung ein Fehlverhalten entdeckt wird, so löst diese eine Kernelmode-Exception[6] aus, welche dann über den Verursacher und das genaue Fehlverhalten Auskunft gibt. Die nachfolgende Tabelle [6.2] gibt Auskunft über das Testziel der möglichen Überprüfungstypen von *Verifier*.

Überprüfungstypen von *Verifier*	
Überprüfungstyp	**Beschreibung**
Spezieller Pool	Test auf Speicherüber- bzw. –unterlauf
IRQL-Überprüfung	Überprüfung, ob IRQL gültig
Geringe Ressourcensimulierung	Test des Treiberverhaltens bei unzureichenden Systemressourcen
Poolnachverfolgung	Test auf nicht freigegebenen Speicher (Memoryleaks)
E/A –Überprüfung	Test der Kommunikationsschnittstelle Treiber – Betriebssystem

Tabelle [6.2]

Nachdem ein oder mehrere Treiber mit *Verifier* zur Überprüfung markiert wurden und das System neu gestartet wurde, kann der Treiber normal verwendet werden. Da nur das systembezogene Verhalten getestet wird und nicht die Funktionalität bezüglich des zugehörigen physischen Geräts, ist es für einen vollständigen Test ausreichend, die Kommunikation mit dem Treiber aus einer Anwendung heraus aufzunehmen, eine E/A-Anforderung an diesen zu stellen (beliebiges IOCTL) und die Kommunikation zu beenden. Tritt während eines solchen Testlaufs keine Kernelmode-Exception auf, so kann davon ausgegangen werden, dass der Treiber aus Systemsicht kein Fehlverhalten zeigt.

[6] Erkennbar durch den bekannten BSoD – BlueScreen of Death, dieser markante blaue Textbildschirm setzt den Benutzer darüber in Kenntnis, dass eine schwerer Fehler entdeckt und darauf hin das System angehalten wurde.

Beim Test der eigentlichen hardwarebezogenen Funktionalität der Treiber wird ebenfalls funktionsorientiert vorgegangen. Dabei werden die einzelnen vom Treiber mit Hilfe der IOCTL-Schnittstelle zur Verfügung gestellten Funktionen sukzessive getestet. Um diese Tests durchführen zu können, wurde ein separates Programm erstellt, mit dessen Hilfe gezielte Anfragen an den jeweils zu testenden Treiber gestellt werden können. Dieses Programm implementiert den unter [Kapitel 5 – Beschreibung der Umsetzung von Hardwarezugriffen im Portierungsergebnis] beschriebenen Mechanismus der Kommunikation mit Gerätetreibern über mit Hilfe der API-Funktion *DeviceIoControl()*. Die in Tabelle [5.6] aufgeführten IOCTL-Codes werden nacheinander getestet[7]. Tabelle [6.3] zeigt die Testsequenzen, denen alle XCTL-Gerätetreiber unterzogen wurden.

IOCTL-Testsequenzen für alle XCTL –Gerätetreiber						
Test-sequenz		**ICOTL-Code**	**Parameter**	**Rückgabe-wert**	**erwarteter Wert**	**Fehlercode **)**
1		IOCTL_DC_ REPORT_ID	<keine>	ULONG	Basisadresse entsprechend Wertebereich *)	0
2	A	IOCTL_DC_ READ_BYTE	BYTE	BYTE	von der Hardware gelesenes Byte	0
	B		BYTE	BYTE	undefiniert	IOCTL – Fehlercode
3	A	IOCTL_DC_ WRITE_BYTE	BYTE	<keiner>	undefiniert	0
	B		BYTE	<keiner>	undefiniert	IOCTL – Fehlercode

A- Parameter Wertebereich*) entsprechen den Treiberressourcen
B- Parameter Wertebereich*) entsprechen nicht den Treiberressourcen
*) gültige Wertebereiche der Treiberressourcen können Tabelle [D.1] entnommen werden
**) der Fehlercode einer IOCTL –Operation wird von *GetLastError()*nach dem Fehlschlagen eines Aufrufs von *DeviceIoControl()* zurückgegeben [Kapitel 5 – Beschreibung der Umsetzung von Hardwarezugriffen im Portierungsergebnis].
Tabelle [6.3]

Die Funktionalität der Treiber wurde bei eingebundener Hardware getestet. Ein Test der Treiber ohne entsprechende Hardware ist hier nicht notwendig, da jeder XCTL-Gerätetreiber aufgrund seiner in der Spezifikation geforderten Hardwareerkennungsfähigkeit überhaupt nur

[7] Ausgenommen davon sind die IOCTL-Codes: IOCTL_DC_SET_WFR_CYCLES und IOCTL_DC_GET_DATA , da diese nur den Gerätetreiber Braun-PSD betreffen.

dann geladen wird, wenn auch tatsächlich die zugehörige Hardware im System installiert ist.
Ein Test der Hardwareerkennungsfähigkeit gestaltet sich sehr einfach, da die Treiber beim Systemstart die Hardwareerkennung durchführen und bei einem Fehlschlagen derselben sofort wieder entladen werden. Ob ein Gerätetreiber erfolgreich geladen wurde, kann u.a. mit dem *Gerätemanager* überprüft werden. Wenn ein Treiber nicht geladen werden konnte, erscheint dort ein gelbes Ausrufezeichen an seinem Eintrag. Damit ergibt sich folgende Tabelle für den Test der Hardwareerkennung. [6.4]

Testsequenzen - Hardwareerkennung der XCTL-Gerätetreiber	
Testsequenz	**Zustand des XCTL -Gerätetreibers nach Systemstart**
A	Geladen
B	Nicht geladen

A- Hardware im System installiert
B- Hardware nicht im System installiert

Tabelle [6.4]

Wie bereits erwähnt, weicht die Schnittstelle des Treibers für den *Braun-PSD* von denen der anderen XCTL-Gerätetreiber ab. Aus diesem Grund wird dieser Treiber über die bereits beschriebenen Tests hinaus geprüft.
Da diese erweiterte Funktionalität nur in einem bestimmten Kontext sinnvoll getestet werden kann, wird aus Gründen der Effizienz auf die portierte Version von XCTL als Testtreiber zurückgegriffen, Testsequenzen dazu siehe Tabelle [6.5].

Testsequenzen – IOCTL -Erweiterung des Braun-PSD Gerätetreibers	
Testsequenz	**erwartetes Verhalten *)**
A	0
B	gleiche Messwerte (mit Toleranz) wie in vergleichbarem Testlauf unter XCTL 16-bit

A- Messwert ohne angeschlossenen Detektorkopf
B- Für diesen Test ist eine vollständige Messung durchzuführen und dann mit der gleichen Messung unter der XCTL 16Bit zu vergleichen. Siehe dazu Regressionstest von XCTL

*) mit Verhalten sind hier die Messwerte des Braun-PSD in XCTL gemeint

Tabelle [6.5]

Leider kann der hier beschriebene Test des Braun-PSD-Treibers nicht mit entsprechenden Testergebnissen belegt werden, da der notwendige Messplatz aufgrund einer Fehlfunktion der Röntgenquelle für einen abschließenden, protokollierten Test nicht mehr zur Verfügung steht. Zwischenzeitlich durchgeführte ad-hoc-Tests der Autoren zeigten jedoch

ein konvergentes Verhalten der Messwerte in Portierungsbasis und Portierungsziel[8].

Test von XCTL

Zur Durchführung des Regressionstests wird das Werkzeug ATOS eingesetzt. Hiermit kann ein automatisierter oberflächenbasierter Test eines Softwaresystems durchgeführt werden. Voraussetzung hierfür ist die Definition von Testsequenzen, die im Wesentlichen aus der Bedienung von Oberflächenelementen des Testgegenstandes bestehen.

Dem definierten Testverfahren entsprechend gliedert sich dieser Testvorgang in zwei Phasen.

(1) Test in der simulierten Umgebung

Ziel dieses Tests ist die Verifizierung der Grundfunktionalität des Portierungsergebnisses. Hierfür werden die Testsequenzen des anerkannten Regressionstests sowohl für die Portierungsbasis als auch das Portierungsergebnis durch ATOS unter Windows 2000 abgearbeitet und die Testergebnisse protokolliert. Der Testgegenstand wird jeweils in der simulierten Umgebung ausgeführt.

Bestandene Testfälle bedürfen keiner weiteren Analyse. Sollte ein Testfall jedoch fehlschlagen, so ergibt sich die Notwendigkeit einer Untersuchung der Ursachen für dieses Testergebnis, da nicht zwangsweise von einem Fehlverhalten des Testgegenstandes ausgegangen werden muss.

Ein Testfall für das Portierungsergebnis gilt unabhängig von seinem Testergebnis als bestanden, wenn der entsprechende Test der Portierungsbasis das gleiche Testresultat erzeugt. Mit anderen Worten: Sollte ein Testfall sowohl für die Ausgangsversion als auch die portierte Version fehlschlagen, so gilt dieser trotzdem als bestanden, da ja nur Fehler, die durch den Portierungsvorgang entstanden sind, aufgezeigt werden müssen.

[8] Bestätigung durch Dr. Hanke möglich

(2) Test der hardwarebezogenen Anwendungsfälle in der Realumgebung

Ziel dieses Tests ist die Verifizierung der Funktionalität der Hardwaresteuerung des Portierungsergebnisses. Die dafür notwendigen Referenzdaten müssen manuell erzeugt werden (siehe Generierung von Referenzdaten). Hierzu wird die Portierungsbasis auf der Testplattform unter Windows 3.11 ausgeführt und die entsprechenden Testsequenzen manuell abgearbeitet. Für die synthetischen Motortests ist es dabei notwendig, die für die einzelnen Testschritte benötigten Zeitspannen zu messen, da diese als Referenzdaten benötigt werden. Alle anderen Referenzdaten werden implizit durch Ausführen der Testsequenzen erzeugt.

Anschließend werden die gleichen Testsequenzen für das Portierungsergebnis unter Windows 2000 durch ATOS ausgeführt. Für die Motortests fließen die Referenzdaten dabei schon in den Ablauf der Testsequenzen ein, so dass der Erfolg der Tests direkt erkennbar ist. Für alle anderen Testfälle muss abschließend ein manueller Vergleich der erzeugten Messdaten mit den Referenzdaten durchgeführt werden.

Die Motivation der synthetischen Motortests liegt in Effizienzüberlegungen begründet. So ist immer mit einer bestimmten Wahrscheinlichkeit davon auszugehen, dass ein Test nicht erfolgreich beendet wird. Das Fehlschlagen eines Motortests an einem Messplatz führt zu einer Dejustierung des Messplatzes, was aufwendig korrigiert werden muss. Um dies zu vermeiden, entschieden sich die Autoren, die entsprechenden Tests mit separaten Testmotoren durchzuführen. Diese sind aus Softwaresicht in ihren Eigenschaften identisch zu den Motoren eines Messplatzes. Teilweise handelt es sich sogar um baugleiche Modelle. Die einzelnen Testfälle werden an jeweils nur einem Motor der jeweiligen Controllerkarte durchgeführt, da hiermit schon die gesamte Funktionalität abgedeckt wird. Ein korrektes Ansprechen (Adressieren) der Motoren wird in einem speziellen Zuordnungstest an einem Messplatz geprüft, bei dem jeder Motor über seinen Namen gezielt angesprochen und bewegt wird. Dieser Test wird manuell und lediglich für das Portierungsergebnis durchgeführt.

Testergebnis

Test der Hardwaretreiber

(1) Test des Verwaltungsfunktionen und des Verhaltens der Treiber im System

Der abschließende Testlauf von *Verifier* wurde ohne Auftreten einer Kernelmode-Exception durchgeführt, so dass davon ausgegangen werden kann, dass die Treiber in dieser Hinsicht korrekt implementiert wurden.

(2) Test der hardwarebezogenen Funktionalität

Der Test der Verarbeitung der grundlegenden IOCTL-Codes mittels des selbst geschriebenen Testprogramms verlief abschließend für alle XCTL-Gerätetreiber erfolgreich. D.h., sowohl das Lesen und Schreiben von Daten (IOCTL_DC_READ_BYTE, IOCTL_DC_WRITE_BYTE) als auch das Auslesen der eingestellten Hardwareadresse (IOCTL_DC_REPORT_ID) konnten ohne auftretende Fehler durchgeführt werden.

(3) Test der Hardwareerkennungsfähigkeit

Alle XCTL-Gerätetreiber werden bei korrekt installierter Hardware ordnungsgemäß geladen bzw. bei nicht korrekt installierter Hardware ordnungsgemäß entladen.

(4) Test der IOCTL -Erweiterungen des Braun-PSD Gerätetreibers

Die ermittelten Messwerte entsprechen insofern den Erwartungen, dass bei nicht angeschlossenem Detektorkopf im Zählerfenster von XCTL der Wert 0 angezeigt wird. Bei angeschlossenem Detektorkopf stimmen die Messwerte der Portierungsbasis und des Portierungsergebnisses weitestgehend überein.

Die grundlegende Funktionalität der Treiber wurde durch obige Testergebnisse verifiziert.

Test von XCTL

(1) Test in der simulierten Umgebung

Entsprechend der Definition, dass ein Testfall als bestanden gewertet wird, wenn sowohl Portierungsbasis als auch Portierungsergebnis das gleiche Testergebnis aufweisen, wurden die Testfälle bis auf eine Ausnahme, in der simulierten Umgebung bestanden. Die Testsequenz *Test_LS.3* schlug für das Portierungsergebnis beim Vergleich der Messwertdateien fehl. Ursache sind allerdings nicht fehlerhafte Messdaten, sondern eine nicht erwartete Anzahl von Messwerten. Die Testsequenz führt einen *Line-Scan* für *Omega*-Positionen von *0* bis *30 Grad* bei einer Schrittweite von *2 Grad* aus. Die erwartetet Anzahl von Messwerten ist entsprechend *16* (30 / 2 + 1). Das ist auch die Anzahl der Messwerte die das Portierungsergebnis liefert. Die ATOS-Referenzdatei erwartet allerdings *49* Messwerte. Das Fehlschlagen der Testsequenz für das Portierungsergebnis ist an diesem Punkt also korrekt. Der in der Portierungsbasis enthaltene Fehler muss im Portierungsergebnis zwischenzeitlich korrigiert worden sein.

Da Letztendlich alle Testfälle als bestanden gewertet werden, kann davon ausgegangen werden, dass die Grundfunktionalität des Portierungsgegenstandes korrekt portiert wurde.

Testsequenz	Win16 (Portierungsbasis)	Win32 (Portierungsergebnis)	Ergebnis
Test_AE.1	OK	OK	OK
Test_AE.2	OK	OK	OK
Test_AJ.1	OK	OK	OK
Test_ARS.1	Fehler	Fehler	OK
Test_ARS.2	Fehler	Fehler	OK
Test_ARS.3	Fehler	Fehler	OK
Test_ARS.PD.1	Fehler	Fehler	OK
Test_AS.1	OK	OK	OK
Test_D0.1	OK	OK	OK
Test_D0.2	OK	OK	OK
Test_DM.1	OK	OK	OK
Test_DM.2	OK	OK	OK
Test_HWB.1	OK	OK	OK
Test_LS.1	OK	OK	OK
Test_LS.2	OK	OK	OK
Test_LS.3	OK	Fehler	OK
Test_LS.PD.1	OK	OK	OK

Test_MJ.1	OK	OK	OK
Test_MJ.2	OK	OK	OK
Test_MJ.3	OK	OK	OK
Test_MJ.4	OK	OK	OK
Test_MJN.1	OK	OK	OK
Test_MJN.2	OK	OK	OK
Test_MJN.3	OK	OK	OK
Test_MJN.4	OK	OK	OK
Test_MJN.5	OK	OK	OK
Test_MJN.6	OK	OK	OK
Test_MS.1	OK	OK	OK
Test_MS.2	OK	OK	OK
Test_MS.3	OK	OK	OK
Test_PD.1	OK	OK	OK
Test_PD.2	OK	OK	OK
Test_PT.1	OK	OK	OK
Test_PT.2	OK	OK	OK
Test_TP.1	OK	OK	OK
Test_TP.2	OK	OK	OK
Test_TP.HWB.PT.1	OK	OK	OK
Test_TPG.1	OK	OK	OK
Test_TPG.2	OK	OK	OK

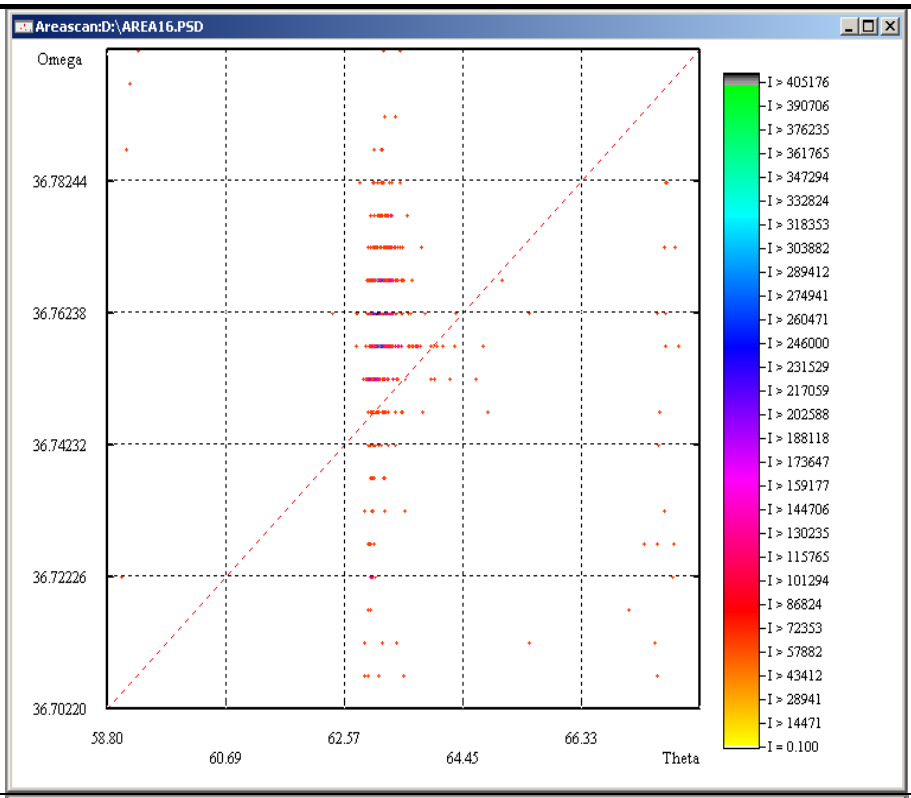
(2) Test der hardwarebezogenen Anwendungsfälle in der Realumgebung

Test_AJ.H.1	
Win16	Die automatische Justage konnte auf dem Testsystem nicht erfolgreich ausgeführt werden, da zum einen ein „Floating Point Error“ auftrat bzw. der Vorgang noch vor dem ersten Durchlauf stehen blieb und nicht terminierte.
Win32	Auf dem Testsystem wurde automatisch ein Maximum bei TL: -46.37, DF: 31.39, CC: -60.75 gefunden.

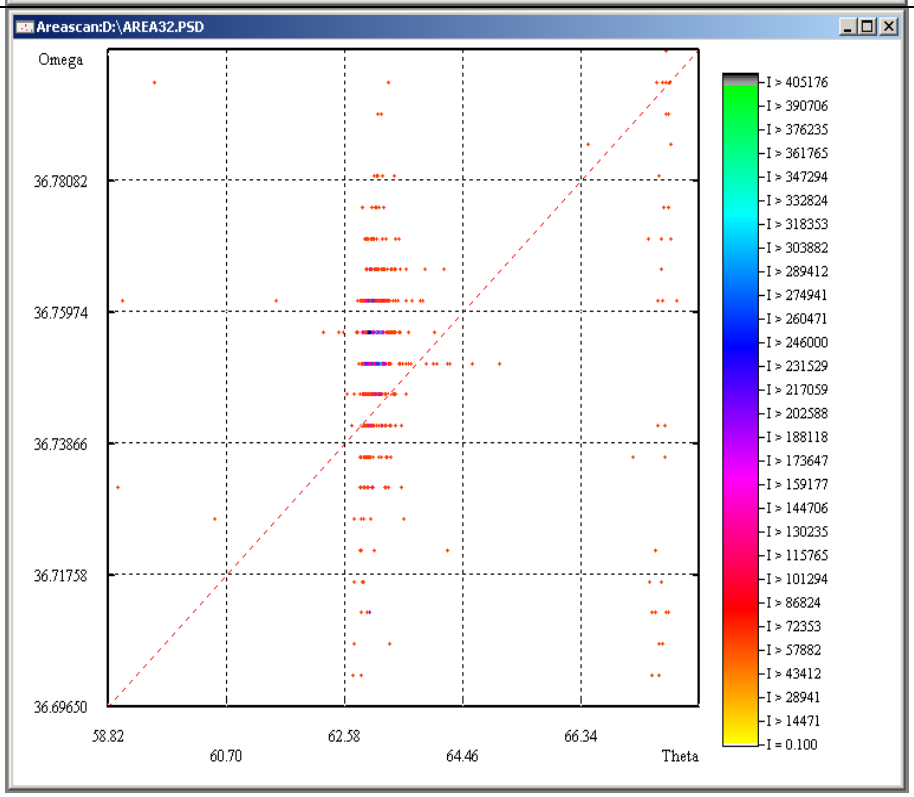
Da der entsprechende Testfall in der Portierungsbasis nicht erfolgreich ausgeführt werden konnte, stehen keine gültigen Referenzdaten zur Verfügung. Allerdings kann der Test trotzdem als bestanden gewertet werden, da nach Abschluss des Vorgangs die Probe tatsächlich in einem relativen Maximum positioniert ist.

Test_ARS.H.1

Win16



Win32



Ein Vergleich der Messwertdateien mit *DataDiff* kann leider nicht erfolgen, da empirische Untersuchungen der Autoren zeigten, dass dieses Werkzeug für die Auswertung von realen Messwerten, die durch nicht quantifizierbare Schwankungen gekennzeichnet sind, einfach zu unflexibel ist. Erschwerend kommt hinzu, dass die Messwerte aufgrund eines Implementierungsfehlers, der schon in der Portierungsbasis enthalten ist, nicht aussagekräftig sind (durch zu große Messwerte werden interne Bereichsüberläufe provoziert). Dieser Fehler der Portierungsbasis ist allgemein bekannt.
Ein visueller Vergleich der grafischen Darstellung der Messwerte lässt allerdings erkennen, dass die Messwerte eine signifikante Ähnlichkeit aufweisen, so dass dieser Testfall als bestanden gewertet werden kann.

Das beschriebene Problem der Vergleichbarkeit der Messwerte tritt im gleichen Maße auch bei den folgenden Anwendungsfällen *Detektornutzung* und *Line-Scan* auf. Auch hier sind die typischen Schwankungen nicht derart quantifizierbar, dass ein automatisierter Vergleich der Messdaten mit *DataDiff* möglich wäre. Aus diesem Grund wird bei der Auswertung der Messdaten des Anwendungsfalls *Line-Scan* wiederum auf einen visuellen Vergleich der grafischen Darstellung der Daten zurückgegriffen. Für die Messwerte der Anwendungsfälle Detektornutzung ist dies nicht so leicht möglich, da existierende Messdaten nicht im Programm nachgeladen und dargestellt werden können. Zur Auswertung der erzeugten Testdaten wird hier auf Methoden der Statistik zurückgegriffen. (Die Messwerte wurden auf ganze Zahlen gerundet.)

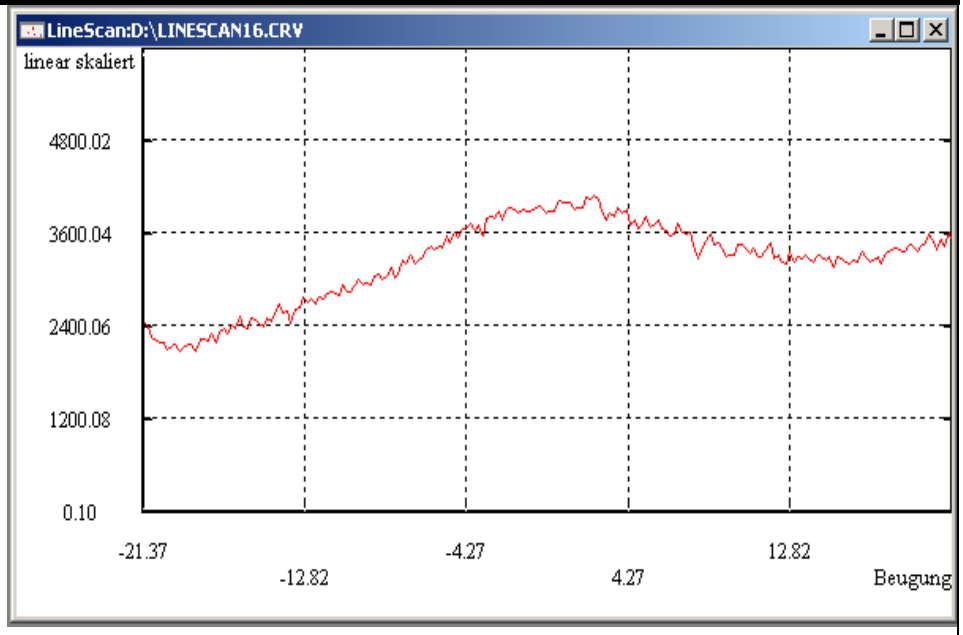
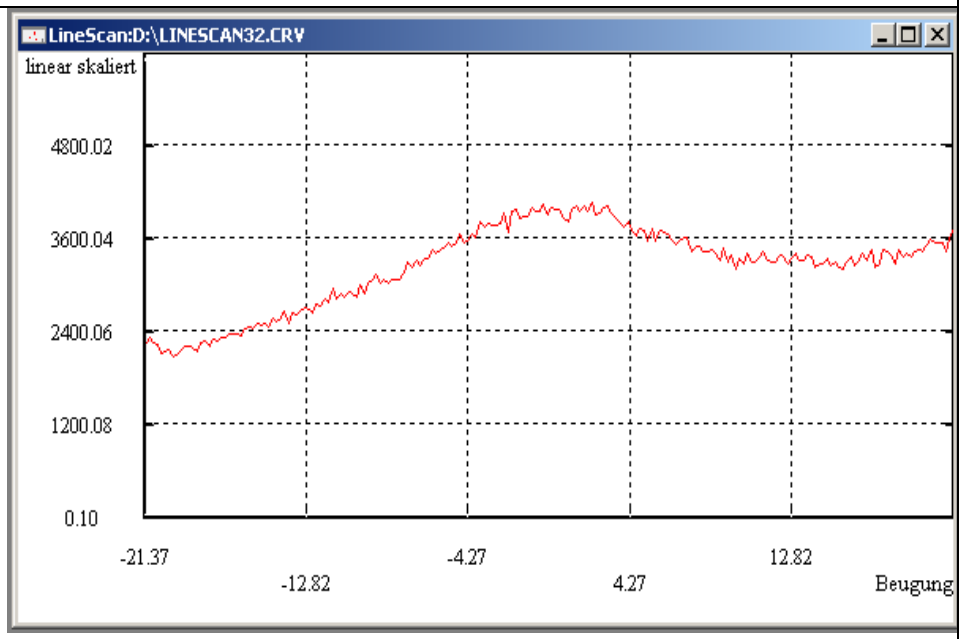
Test_D0.H.1					
Zeit	Impulse		Win16	Win32	Standard-ab-weichung
0,5	20000	minimaler Wert	19746	20357	
		maximaler Wert	20585	20952	
		arithmetisches Mittel	20273	20701	≈1,05%
1,0	20000	minimaler Wert	20078	20430	
		maximaler Wert	20465	21099	
		arithmetisches Mittel	20273	20277	≈0,00%
1,0	30000	minimaler Wert	20065	20353	
		maximaler Wert	20478	20971	
		arithmetisches Mittel	20259	20735	≈1,16%

Die Standardabweichung der Messwerte von Portierungsbasis und Portierungsziel liegt im Bereich von einem Prozent, so dass dieser Test als bestanden gewertet werden kann.

Test_D0.H.2					
Zeit	Impulse		Win16	Win32	Standard-ab-weichung
0,5	20000	minimaler Wert	61489	60670	
		maximaler Wert	80920	70994	
		arithmetisches Mittel	65755	65273	≈0,73%
1,0	20000	minimaler Wert	32812	32904	
		maximaler Wert	47415	38591	
		arithmetisches Mittel	36212	35025	≈3,20%
1,0	30000	minimaler Wert	52547	52758	
		maximaler Wert	57295	58068	
		arithmetisches Mittel	54909	55082	≈0,16%

Die Standardabweichung der Messwerte von Portierungsbasis und Portierungsziel liegt im Mittel bei 1,4 Prozent (und nie über vier Prozent), so dass dieser Test als bestanden gewertet werden kann.

Der Testfall *Test_D1.H.1* kann aufgrund des Ausfalls der Röntgenquelle nicht mit Testdaten belegt werden. Die schon angesprochenen früheren ad-hoc-Tests der Autoren (z.B. der Test der IOCTL-Erweiterung des Braun-PSD-Treibers) zeigten allerdings ein konvergentes Verhalten der Messwerte in Portierungsbasis und Portierungsziel.

Test_LS.H.1	
Win16	
Win32	

Die beiden Messwertkurven zeigen eine signifikante Ähnlichkeit, so dass der Test als bestanden gewertet werden kann.

Die Ergebnisse der hardwarebezogenen Testfälle *Manuelle Justage* und *Motorsteuerung* werden in der folgenden Tabelle zusammengefasst.

Test_MJ.H.x.x/Test_MS.H.x.x				
	C-812 (Test_xx.x.1)		C-832 (Test_xx.x.2)	
	Win16	Win32	Win16	Win32
Direktbetrieb (Test_MJ.1.x)	OK	OK	OK	OK
Schrittbetrieb (Test_MJ.2.x)	OK	OK	OK	OK
Fahrbetrieb (Test_MJ.3.x)	OK	OK	OK	OK
Direkte Steuerung (Test_MS.1.x)	OK	OK	OK	OK
Referenzpunktlauf (Test_MS.2.x)	OK	OK	- *)	-*)
Motoroptimierung (Test_MS.3.x)	OK	OK	OK	OK

*) der Referenzpunktlauf konnte nicht durchgeführt werden, da der entsprechende Testmotor nicht über Endlagenschalter verfügt

Die Test können allesamt als bestanden gewertet werden. Auch der oben erwähnte Test der Motoradressierung zeigt, dass die Motoren korrekt ihren in der Konfigurationsdatei von XCTL definierten Bezeichnern zugeordnet und angesprochen werden.

Während der Motortests zeigte sich, dass die Motorcontrollerkarte *C-812* nicht in Computern betrieben werden kann, die auf dem *Intel BX*-Chipsatz für *Pentium II/III* basieren. Hierbei handelt es sich um eine hardwareseitige Inkompatibilität, die nicht im Zusammenhang mit der Portierung steht.

Zusammenfassung

Sowohl der Regressionstest in der simulierten Hardwareumgebung als auch die hardwarebezogenen Tests in der Realumgebung haben gezeigt, dass das Portierungsergebnis im gleichen Maße funktioniert wie die Portierungsbasis. Die Portierung kann also als erfolgreich bezeichnet werden.

7. Fehlerbereinigung

Dieser Abschnitt beschreibt die Fehler, die sich erst nach der Portierung durch Fehlverhalten von *XCTL* äußerten, aber bereits in der Ausgangsversion bestanden. Erst durch die strengeren Verwaltungsmechanismen der Ausführungsumgebung (z.B. Speicherverwaltung) von Windows 2000 konnten diese Fehler zutage treten. Die Ursachen dieser Fehler und, sofern möglich, ihre Bereinigung werden hier beschrieben.

Timingprobleme

Ein weitreichendes Problem sind die im Programm verwendeten Warteschleifen. Wenn nach bestimmten I/O-Operationen eine gewisse Zeit zu warten ist, so wurde dies bislang durch Schleifenkonstrukte mit einem Zähler realisiert. Wenn der Zähler einen bestimmten Wert erreicht, wird die Warteschleife beendet. Diese Vorgehensweise ist in älteren Programmen weit verbreitet, was damit zusammenhängt, dass es an adäquaten Bibliotheksfunktionen mangelte, die in der Lage sind, eine bestimmte Zeitspanne zu warten. Ein solches Verfahren hat den gravierenden Nachteil, dass eine Warteschleife in ihrem Zeitverhalten von der verwendeten Hardware abhängig ist. Auf einem schnelleren System wird die Schleife einfach schneller abgearbeitet. Das bedeutet, dass Wartezeiten in der Kommunikation mit der Hardware[1] zu kurz ausfallen und somit die Kommunikation nicht mehr funktionieren kann. In der portierten Version machte sich dies sofort an vielen Stellen bemerkbar. Als Beispiele dafür seien zwei charakteristische Reaktionen genannt:

(1) Der Detektor vom Typ Braun-PSD kann nicht initialisiert werden, da entweder das Hochladen der Firmware-Datei *ASA23.HEX* fehlschlägt oder der direkt folgenden Controllerbefehl nicht korrket abgearbeitet wird.

(2) Ein Motor vom Typ C-812 kann nicht initialisiert werden.

Die Untersuchung des beschriebenen Verhaltens ergibt, dass die globalen Funktionen *Delaytime()* und *Delay()* verwendet werden.

[1] indirekt über Gerätetreiber

Beide warten eine bestimmte Zeit. Die Funktion *DelayTime()* bereitete nach der Portierung keine Probleme, da sie eine absolute Zeitspanne wartet. Ihr gesamter Funktionsrumpf wurde durch einen einzigen Aufruf von *Sleep()* ersetzt, da dies den aufrufenden Thread für die Wartezeit suspendiert und nicht unnötig Prozessressourcen verschwendet werden.

Das eigentliche Timingproblem liegt hauptsächlich in der Funktion *Delay()*. Hier findet sich ein Schleifenkonstrukt, welches einen Zähler aufaddiert. Wenn der zu Beginn aus übergebenem Parameter und einem Korrekturfaktor errechnete Wert erreicht ist, kehrt *Delay()* zurück. Tabelle [7.1] gibt Auskunft über die Funktionen, welche *Delay()* verwenden. Da keinerlei Dokumentation zu dieser Funktion existiert, ist es sehr schwer, Aussagen über den Zeitraum zu treffen, der bei einem spezifischen Parameterwert gewartet werden soll. Dies jedoch ist die Grundlage einer adäquaten Portierung. Auch die Anwendung von *Delay()* in den entsprechenden Funktionen lässt keine Rückschlüsse auf das notwendige Zeitverhalten zu. Um das Problem der Hardwarabhängigkeit der Kommunikation zu beseitigen, wird die Funktion *Delay()* so reimplementiert, dass dort eine absolute Zeitspannen gewartet wird.

Prinzipiell gäbe es keine Nachteile für die Kommunikation, wenn nach jedem Hardwarezugriff mindestens *1 ms* gewartet würde, damit könnte *Sleep()* auch hier in *Delay()* zum Einsatz kommen. Allerdings würden sich aufgrund der Implementation der Hardwarekommunikation diese *Sleep()*-Aufrufe derart aufsummieren, dass es teilweise zu extremen Wartezeiten kommt. Ein Beispiel dafür sind die Dialoge der *Motoroptimierung*, die nach dem Versuch einer solchen Modifikation nicht mehr funktionierten, da die Antwortzeiten der von den Dialogen aufgerufenen Funktionen nun zu groß waren. Leider existieren keine Bibliotheksfunktionen, die eine höhere zeitliche Auflösung bieten als 1 ms. Recherchen haben ergeben, dass es mit Hilfe sogenannter *High Resolution Performance Counter* möglich ist, eine zeitliche Auflösung von bis zu 1µs zu erreichen. *High Resolution Performance Counter* sind spezielle Hardwareeinheiten, die pro Sekunde eine garantierte Anzahl von Zähleinheiten zählen. Mit dem Wissen, wie viele Zähleinheiten pro Sekunde gezählt werden und wie viele seit Beginn des zu messenden Zeitraumes gezählt wurden, kann eine hochauflösende Funktion realisiert werden. Nachteilig ist, dass zum einen nicht auf jedem System diese Zähler zur Verfügung stehen und zum anderen, dass diese, wenn sie denn vorhanden sind, periodisch abgefragt werden müssen.

Die benötigte hochauflösende Wartefunktion wurde wie folgt realisiert:

(1) Eine Initialisierungsfunktion *InitializeDelay()* errechnet mit Hilfe der Bibliotheksfunktion *QueryPerformanceFrequency()* und der globalen Variable *dDelayMin* einen Zählerfaktor, der in der modifizierten Funktion *Delay()* verwendet wird. Dabei gibt *dDelayMin* die Zeit an, die ein Aufruf *Delay(1)* in µs dauern soll. Die Funktion *InitializeDelay()* ist einmal vor der ersten Verwendung von *Delay()* aufzurufen. Sollten keine *High Resolution Performance Counter* existieren, dann wird mit einer Meldung darauf hingewiesen, dass es zu Problemen kommen kann.

(2) Die Funktion *Delay()* wird modifiziert. Mit Hilfe der Bibliotheksfunktion *QueryPerformanceCounter()* wird bei Funktionseintritt der Stand der *High Resolution Performance Counter* ermittelt und dann mit dem in *InitializeDelay()* berechneten Faktor und dem übergebenen Parameter *count* der Zielwert errechnet. In einer Schleife wird nun der Stand der *High Resolution Performance Counter* solange ermittelt und mit dem berechneten Zielwert verglichen, bis dieser erreicht ist.
Wenn auf dem System keine *High Resolution Performance Counter* existieren, so verwendet *Delay()* die Funktion *Sleep()*, was aber, wie bereits beschrieben, zu Fehlern führen kann.

Da sämtliche zukünftig in der Physik verwendeten PC Systeme Pentium oder entsprechend vergleichbare Prozessoren enthalten, ist davon auszugehen, dass diese Systeme alle *High Resolution Performance Counter* besitzen.
An dieser Stelle ist unbedingt anzumerken, dass die derzeit in XCTL verwendete minimale Zeiteinheit von 100µs einen Kompromiss darstellt, der in empirischen Untersuchungen ermittelt wurde. Aufgrund der aktuellen Architektur der Motorkomponenten und Teilen der Detektorkomponenten sowie einiger Oberflächenkomponenten (Motorenoptimierung) war die beschriebene Lösung die geeignete Alternative, ohne größere Änderungen an der Architektur dieser Komponenten durchführen zu müssen. Jedoch sollten diese Komponenten unbedingt einer Überarbeitung unterzogen werden. Näheres dazu im *Kapitel 9 – Ausblick*.

Funktion *Delay()*, verwendet von
TScan::Create(), TAreaScan::Create(), TCalibratePsdDlg::Dlg_OnInit(), TC_812ISA::ExecuteCmd(), TC_812Controller::ExecuteCmd(), TC_812ISA::GetChar(), TC_812Controller::GetChar(), TC_832::IsIndexArrived(), TBraunPsd::Look_till_BaseAddr1(), TStoePsd::PsdInit(), TStoePsd::PsdRead(), TStoePsd::PsdStart(), TStoePsd::PsdStop(), TC_812ISA::PutChar(), TC_812Controller::PutChar(), TC_832Controller::PutWord(),

TGenericController::ReadData(), TGenericController::ReadStatus(), TGenericController::SelectChip(), TC_812::StopDrive(), TGenericController::WriteCmd(), and TGenericController::WriteData()

Tabelle [7.1]

Fehler in Motors.dll

Im Zuge der Implementierung der Treiberkommunikation fiel auf, dass die ursprüngliche Motorarchitektur völlig unzureichend ist. Zum einen konnte nie mehr als eine Motorkarte je Typ gleichzeitig verwendet werden. Ursache dafür ist der schon erwähnte semantische Fehler, der sich z.B. in statischen Membern der Motorklassen äußert. Dieser Fehler wurde durch die Einführung einer weiteren Abstraktionsschicht behoben. Weitere Details dazu sind im *Kapitel 5 – Umstellung der direkten Hardwarezugriffe auf Treiberzugriffe* beschrieben.

Fehler in TBraunPsd

Bei der Untersuchung der Klasse *TBraunPSD* im Rahmen der Analyse von bekanntem und neuem Fehlverhalten wurden einige schwerwiegende Fehler sowie Hinweise auf mögliche Fehler gefunden, die, soweit möglich und im Rahmen der Portierung erforderlich, behoben wurden.

Methode **TBraunPsd::PsdInit()**
Folgendes Codefragment aus dieser Methode ist semantisch falsch:

```
...
if ( bDebug )
{
      lpfnSetTimeout(10000);
      ResetDelayTime( );
}
else
{
      SetInfo( "Debug BPsd" );
      return R_OK;
}
...
```

Die globale Variable *bDebug* ist gesetzt, wenn es sich um einen Debuglauf handelt. In jeder anderen Funktion wird bei gesetztem *bDebug* die Funktion *SetInfo()* aufgerufen und abschließend mit *return R_OK* zurückgekehrt. In der Methode *PsdInit()* ist es genau umgekehrt: Da *bDebug* üblicherweise nicht gesetzt ist, wird die Funktion vorzeitig beendet. Dieses Verhalten ist die Ursache dafür, dass die 16-bit XCTL-Version nicht ohne die Unterstützung eines speziellen Programms *Asa.exe*[2] auf den Detektor *Braun-PSD* zugreifen kann. Bislang musste dieses Programm vor *XCTL* gestartet werden, um durch Laden der Firmware *ASA23.HEX* den Detektor zu initialisieren. Die Korrektur dieses Fehlers wurde mit folgendem Code realisiert:

```
...
if ( bDebug )
{
      SetInfo( "Debug BPsd" );
      return R_OK;
}
Hardware->SetTimeout(10000); // portiertes SetTimeout()
ResetDelayTime( );
...
```

Dadurch ist *XCTL* nun in der Lage, die Datei *ASA23.HEX* eigenständig auf den *Braun-PSD*-Controller zu laden und diesen zu initialisieren.

Methode **TBraunPsd::LoadHexFile()**
Mit der Anweisung:

```
ifstream myfile( Filename );
```

legt diese Methode eine Datei namens *Filename* an, sollte diese nicht existieren. Das ist sicher kein gewünschtes Verhalten, da es sich bei der Datei um die Firmware der Detektorkarte handelt. Richtig muss es heißen:

```
ifstream myfile(Filename,ios::in | ios::nocreate);
```

[2] Dabei handelt es sich um ein vom Hersteller des Detektors geliefertes Programm, welches für Windows 3.11 erstellt wurde und der Steuerung des Detektors dient.

Weiterhin wurde folgendes Codefragment als überflüssig identifiziert:

```
...
if ( bDebug )
{
      SetInfo( "Debug BPsd" );
      return R_OK;
}
...
```

Die Funktion *TBraun::LoadHexFile()* wird ausschließlich von *TBraunPSD::PsdInit()* aufgerufen und zwar nur dann, wenn *bDebug* nicht gesetzt ist, wird eine erneute Überprüfung von *bDebug* an dieser Stelle vollkommen überflüssig. Dieses Codefragment wurde entfernt.

Methode **TBraunPsd::PsdInit()**
Versuche mit der portierten Version von XCTL haben ergeben, dass nach dem Hochladen der Firmware auf den *Braun-PSD*-Controller Kommunikationsprobleme auftraten. Diese konnten durch das Einfügen eines entsprechenden Wartezykluses eliminiert werden. Dieser Fehler machte sich unter der 16-bit Version nicht bemerkbar, da dort die Firmware nicht von *XCTL* auf den Controller geladen wurde. Folgende Änderung wurde durchgeführt:

```
...
LoadHexFile()
Sleep(1000)      // neu eingefügter Wartezyklus von 1s
...
```

Methode **TBraunPsd::PsdReadOut()**
Diese Methode ist für Transfer und Akkumulation der Messwerte des Braun PSD zuständig. Jedoch ist der Umgang mit den vom Braun PSD Controller übermittelten Kanälen falsch. Die Unterlagen [Braun 94] belegen, das die ersten 16 übermittelten Kanäle (Sonderkanäle) keine Messwerte enthalten, sondern spezielle Daten, die der Steuerung und Kommunikation dienen. Das bedeutet also, dass ein Offset von 16 Kanälen existiert, der bei der Auswertung der Messwerte berücksichtigt werden muss. Dies jedoch war nicht der Fall, wie der Vergleich von ursprünglichem und korrigiertem Codefragment zeigt:

```
...
memcpy( (LPSTR) lpdwCountBuf, lpReadBufOffset,
  GetChannelNumber() * sizeof(long));
for ( idx = 0;idx <= GetChannelNumber();idx++ )
{
...
}
```

Korrektur:

```
...
memcpy( (LPSTR) lpdwCountBuf, lpReadBufOffset,
(GetChannelNumber()-16)* sizeof(long));
for ( idx = 0;idx < GetChannelNumber()-16;idx++ )
{
...
}
```

Bei der Korrektur wird von dem Wert den *GetChannelNumber()* zurückliefert und der Offset von 16 Kanälen abgezogen, um die Netto-Kanalzahl zu erhalten, was der Anzahl der tatsächlichen Messwertkanäle entspricht. Auch ist der Vergleichsoperator '<=' falsch, da hierdurch definitiv ein Durchlauf zu viel erfolgt und es zu einer Speicherzugriffsverletzung kommt.

Abschließend zur Fehleranalyse des *Braun-PSD* sei erwähnt, dass die Unterlagen zur Programmierung des Controllers [Braun 94] einen gravierenden Fehler enthalten. Die Beispielimplementation der Funktion *GetData()* ist falsch. Diese Funktion wurde unter Win16 von der Bibliothek *ASA.DLL* zur Verfügung gestellt. Unter Win32 wurde der Gerätetreiber des *Braun-PSD* so konzipiert und realisiert, dass dieser jene DLL ersetzt (siehe Kapitel 5 – Spezifikation der Treiber). Dafür war unter anderem auch die Implementation der Funktion *GetData()* nötig. Bei der Entwicklung dieses Treibers fiel dann folgender Fehler in der Dokumentation auf: Die Beispielimplementation von *GetData()* behauptet, dass in der Hauptschleife der Funktion je Durchgang ein *DWord* eingelesen wird, tatsächlich wird jedoch ein *Word* eingelesen. Mit Hinblick auf die Weiterentwicklung von *XCTL* und der Gerätetreiber kann dieser Hinweis durchaus wichtig werden.

Methode **TDC_Drive::Initialize(void)**
Bei dieser Methode fiel auf, dass sie in jedem Falle mit *R_OK* zurückkehrt, auch wenn *CheckBoardOk()* fehlschlägt. Dieser Fehler wurde korrigiert

durch Anpassung des Rückkehrcodes an entsprechender Stelle im Quelltext (*return FALSE;*).

Fehler in TC_812

Methode **TC_812ISA::CheckBoardOk()**

Der in dieser Methode ausgeführte Aufruf von *retval = ExecuteCmd(buf)* wurde nicht auf Erfolg getestet. Dies wurde durch eine entsprechende Auswertung von *retval* korrigiert.

Methode **TC_812:: ExecuteCmd()**
Der Motorcontroller C-812 sendet nach dem Hochfahren des Systems unaufgefordert einen Versionsstring. Damit die Kommunikation mit der Karte funktionieren kann, muss dieser erst vollständig vom Controller gelesen werden. Dabei reicht die in dieser Funktion zu diesem Zweck implementierte Schleife mit 50 Durchläufen nicht aus, was dazu führt, dass die Kommunikation danach nicht synchronisiert mit dem Controller beginnt. Dieses Fehlverhalten konnte erst nach der Korrektur von *TC_812ISA::CheckBoardOk()* auftreten. Der Aufruf von *ExecuteCmd()* schlug vorher immer fehl, da 50 Durchläufe nicht reichen die Zeichenkette vollständig aus dem Kommunikationspuffer zu entfernen. Der Aufruferfolg wurde aber in *CheckBoardOk()* (was immer als erstes aufgerufen wird) nicht ausgewertet. Der zweite Aufruf in *CheckBoardOk()* mit :

```
...
strcpy( buf, "EF\r" );
retval = ExecuteCmd( buf );
...
```

wiederum ist nun in der Lage, die ca. 20 verbliebenen Zeichen des Versionsstrings zu lesen, ohne dass die 50 Durchläufe ausgeschöpft werden müssen, daher an dieser Stelle kein Fehler.

Entsprechend wurde der Fehler korrigiert, indem die Schleifendurchläufe auf 250 erhöht wurden.

Methode **TC_812ISA::PutChar()**
Im Rahmen der Anpassung der *Delay()* Funktion wurden die hier verwendeten *for*-Schleifen modifiziert. Mit einer Anzahl von 30.000 Durchläufen waren diese definitiv überdimensioniert für die neue *Delay()* Funktion.

Weitere Fehler

Die im Folgenden beschriebenen Fehler sind bei routinemäßigen Tests während der Portierung aufgefallen.

Dialog: Zählerfenster → Einstellungen Gerät
Die Auswahl von Detektoren mit Hilfe der Combobox für die Auswahl eines Detektors ist fehlerbehaftet. Wenn in der Konfigurationsdatei zwei Detektoren existieren, die sich im Identifikationsstring nur in den letzten Zeichen unterscheiden, so werden zwar beide korrekt angezeigt, jedoch wird, egal welcher von beiden angeklickt wurde, immer nur einer der beiden tatsächlich ausgewählt. Dieser Fehler wurde bislang noch nicht korrigiert. Vermutlich wird nicht der komplette Identifikationsstring bei der Auswahl überprüft, sondern nur ein Teil der Zeichenkette.

hardware.ini: Motoreinstellungen
Unter bestimmten Umständen (z.B. Programmabsturz, manueller Eingriff) kann es vorkommen, dass Einträge in den Motorsektionen inkonsistent und falsch sind. Eine solche inkorrekte Konfigurationsdatei führt zu teilweise schwerwiegendem Fehlverhalten von *XCTL*. Dies gilt sowohl für die eigentliche Motorsteuerung als auch für die Darstellung des Motorzustandes. *XCTL* kann eine inkonsistente Konfigurationsdatei nicht erkennen. Für die Überarbeitung der Motorenkomponente sollte vorgesehen werden, diese potenzielle Fehlerquelle zu tilgen.

Statuszeile

Die Statuszeile von XCTL, die primär dazu dient, den Zustand der Anwendung anzuzeigen, ist in ihrer bisherigen Form nicht mehr voll funktionsfähig. Zweierlei Fehlverhalten war unter der portierten Version von XCTL zu beobachten:

(1) Das größte Statusfeld auf der rechten Seite der Statuszeile wurde nach der Darstellung einer Zeichenfolge nicht gelöscht, bevor eine neue in diesem Feld angezeigt wurde. Dies hatte zur Folge, dass eine kurze Zeichenfolge nach einer langen diese nicht vollständig überdeckte und somit die resultierende Zeichenfolge eine Mischung aus beiden war, siehe Abbildung [7.1]. Ein Effekt, der durchaus zum Bereich mangelhafter *Usebility* gezählt werden kann.

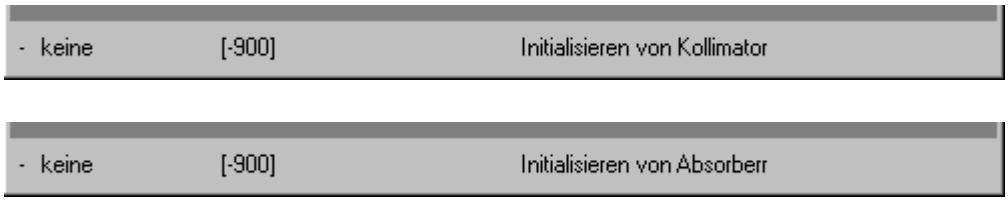


Abb.[7.1] zeigt den unter (1) beschriebenen Effekt während der Initialisierungsphase von XCTL: vom vorhergehenden Text blieb das "r" von Kollimator stehen, so das der Eindruck falscher Orthographie im Begriff "Absorberr" entsteht.

(2) Der Windowsdesktop ist in seiner Farbgestaltung frei variierbar. Bei bestimmten Farbkombinationen enthielt die Statuszeile kein einziges lesbares Element, siehe Abbildung [7.2]

Abb.[7.2] zeigt die fehlerhafte Darstellung der bitmapbasierten Statuszeile von XCTL nach der Portierung bei Verwendung eines vom Standard abweichenden Desktopfarbschemas

Die Ursache für dieses Fehlverhalten liegt in der Implementation der Statuszeile als Bitmapelement. D.h., es dienen Bitmaps dazu, die einzelnen Felder der Statuszeile zu bilden. Diese Bitmaps werden zum einen aus entsprechenden Dateien geladen und zum anderen mit Hilfe der API-Funktion *TextOut()* zur Laufzeit erzeugt. Auf diese Weise wurde eine in jeder Hinsicht statische Statuszeile angelegt, die weder an unterschiedliche Bildschirmauflösungen noch an verschiedenste Farbvariationen des Desktops anpassbar war.
Die Behebung dieses Fehlverhaltens hätte im Prinzip durch Korrekturen der betroffenen Codeteile erreicht werden können, jedoch ist die Implementation einer Statuszeile mittels Bitmaps recht komplex und keineswegs mehr als zeitgemäß zu bezeichnen. Microsoft hat für diesen

Zweck unter Win32 einen eigenen Mechanismus integriert, der folgende Vorteile gegenüber der Bitmapvariante hat:

- in Anzahl, Größe und Layout freikonfigurierbare Statusfelder
- Umkonfiguration der Statusfelder zur Laufzeit möglich
- automatische Anpassung an Änderungen von Bildschirmauflösung und Desktopeinstellungen (Farbe)
- Änderung der Inhalte der Statusfelder mit Hilfe des Fensternachichtensystems von Windows (z.B.: SB_SETTEXT via API-Funktion *SendMessage()*)
- keine zusätzlichen Bitmapdateien benötigt
- Vereinfachung des Quelltextes = weniger potenzielle Fehlerquellen
- Umsetzung des Look-and-Feel der Windowsoberfläche in XCTL

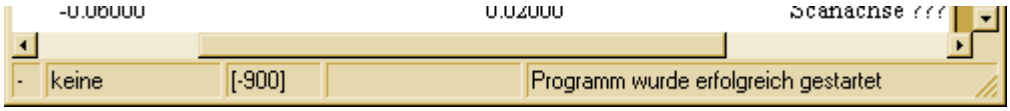


Abb. [7.3] zeigt die neue Statuszeile von XCTL

Die für die Umsetzung einer solchen Win32 konformen Statuszeile in XCTL erforderlichen Schritte sind in den folgenden Tabellen zusammengefasst.

Änderungen in evrythng.h			
Bereich	**Änderung**		**Bemerkung**
Klasse TMain	von	void BlastStatusLine(HWND, HDC, int);	
	nach	void BlastStatusLine(HWND, int);	
	hinzufügen	void CreateStatusBar(HWND);	Methodendeklaration
	hinzufügen	HWND hwndStatusBar;	private Member

Änderungen in m_main.cpp			
Bereich	**Änderung**		**Bemerkung**
TMain::TMain()	hinzufügen	hwndStatusBar(NULL)	Initialisierungs-liste des Konstruktors erweitern

TMain::Create()	löschen	if (StatusLineHeight == 25)
		  bmpStatusLine = LoadBitmap(…);
		else
		  bmpStatusLine = LoadBitmap(…);
		if (!hWndClient)
		  return FALSE;	

	hinzufügen	Main.CreateStatusBar(hWnd);	

	ändern	const int WatchStart = 5;
		const int TargetStart = 15;
		const int ReflectionStart = 105;
		const int FlagStart = 155;
		const int TextStart = 255;
			die vorhandenen Konstanten um 5 erhöhen

TMain ::SetWatchIndicator (long st)	ändern	Inhalte der *case*-Zweige durch char[0]="*" ersetzen, wobei * das entsprechende Zeichen des ursprünglichen *case*-Zweiges ist	komplett leeren bis auf die *switch*-Anweisung
	hinzufügen	SendMessage(hwndStatusBar, SB_SETTEXT, (WPARAM) 1 , (LPARAM) chr);	
void TMain::CreateStatusBar (HWND hWnd)	hinzufügen	CommonControl Dll laden für StatusBar, Fenster erzeugen (StatusBar anlegen) StatusBar unterteilen in 5 Bereiche	neue Funktion implementieren siehe Quelltext (M_Main.cpp)
TMain::DoPaint (HWND hWindow)	reimplemen-tieren	siehe Quelltext (M_Main.cpp)	gesamten Rumpf ersetzen
TMain::DoSize (HWND hWindow)	reimplemen-tieren	siehe Quelltext (M_Main.cpp)	gesamten Rumpf ersetzen
BlastStatusLine (HWND hWindow, int StatusId)	reimplemen-tieren	siehe Quelltext (M_Main.cpp)	gesamten Rumpf ersetzen
TMain::DrawStatus	löschen	Variable hDC und sämtliche mit ihr in Bezug stehenden Anweisungen	

8. Zusammenfassung

Bewertung des Portierungsergebnisses

Der im letzten Kapitel beschriebene umfangreiche, diversifizierende Regressionstest hat bewiesen, dass XCTL erfolgreich nach Windows 2000 und Visual C++ portiert wurde. Dabei wurde die in den theoretischen Betrachtungen geforderte Erhaltung der Funktionalität sicherstellt.
Änderungen an den Quellen wurden hauptsächlich in den Bereichen der Hardwarekommunikation durchgeführt. Damit einhergehend wurden Änderungen an der Architektur der Gerätesteuerung nötig. In diesem Zusammenhang sei erwähnt, dass mit dieser Änderung die Funktionalität sogar erweitert wurde. Während der semantische Fehler der Motorkomponente die Nutzung von mehr als einer Motorkarte des gleichen Typs zur selben Zeit in der Portierungsbasis verhindert hat, so ist XCTL durch Einführung der Treiberarchitektur und Beseitigung dieses semantischen Fehlers nun in der Lage, mit einer theoretisch beliebigen Anzahl von Motorkarten desselben Typs gleichzeitig zu arbeiten. Weniger umfangreiche Eingriffe fanden an der Ablaufsteuerung und der Oberfläche statt.
Im Verlauf der Arbeiten an den Quellen wurden noch weitere Fehler gefunden und behoben, die erst durch die Portierung in Erscheinung treten konnten. Ursache dafür ist der, softwaretechnisch gesehen positiv zu bewertende, strengere Umgang des Zielsystems mit dem Portierungsgegenstand in den Bereichen Syntaxprüfung (Rahmenbedingung Entwicklungsumgebung) und Speicher-/Ressourcenverwaltung (Rahmenbedingung Softwareumgebung/ Betriebssystem).
XCTL ist in vollem Umfang unter dem Zielsystem einsetzbar. Noch bestehende Probleme sind dokumentiert (Fehler: Messdaten Braun-PSD). Dringende und mögliche Entwicklungsschritte für XCTL werden im anschließenden Kapitel 9 *Ausblick* angeregt.

Zusammenfassen des Vorgehensmodells

Abschließend soll das in den theoretischen Grundlagen erörterte Vorgehensmodell noch einmal aufgegriffen werden. Dabei fließen die bei der Anwendung dieses Modells gewonnenen Erfahrungen in Form einer kritischen Auseinandersetzung und einer Aufwandsdarstellung ein.

(1) Analyse der Ausgangssituation und Bestimmen der Ausprägung der Rahmenbedingungen

Der erste Schritt des Modells erwies sich als sehr aufwändig. Im Wesentlichen werden hierbei fast ausschließlich Dokumentationen der Host- und Zielsysteme bezüglich der zu untersuchenden Rahmenbedingungen (Entwicklungs-/Softwareumgebung) durchgearbeitet. Wie bereits in *Kapitel 3 – Analyse Entwicklungsumgebung* erwähnt, stellte sich dies besonders bei der Rahmenbedingung Entwicklungsumgebung als sehr aufwändig dar, weswegen auf einen Teil der Analyse zu Gunsten der im folgenden Schritt der Adaption angewendeten Bottom-Up Methode verzichtet wurde. Die Rahmenbedingung Softwareumgebung ließ sich hingegen relativ problemlos untersuchen.

(2) Adaption bezüglich der Rahmenbedingung Entwicklungsumgebung

Dieser Schritt wurde in Folge des hohen Aufwandes für eine vollständige Analyse der Rahmenbedingung *Entwicklungsumgebung* dahingehend abgeändert, dass die notwendigen Schritte *Analyse* und *Adaption* durch anwenden der Bottom-Up-Methode der Portierung verschmolzen wurden.
Diese Vorgehensweise bot sich an, da davon auszugehen war, dass der Portierungsgegenstand nicht alle Unterschiede der Entwicklungsumgebungen (die in einer vollständigen Analyse ermittelt worden wären) tangiert. Die Bottom-Up-Methode der Portierung basiert auf der Korrektheitsanalyse von Compiler und Linker der Zielumgebung, welche jedes unkorrekte Element der Quellen beim Übersetzungsvorgang anzeigen. Sukzessive werden so alle spezifischen Sprachelemente des Hostsystems eliminiert.

An dieser Stelle ist jedoch zu bemerken, dass die vorgestellte Bottom-Up-Methode der Portierung ihre Grenzen hat. Unter Umständen kann es dazu kommen, dass die vom Compiler des Zielsystems angegebenen Fehlerstellen nicht in den Quellen des Portierungsgegenstandes liegen. In einem solchen Fall wird dann der Fehler z.B. in den Headerdateien der Bibliotheken der Zielentwicklungsumgebung angezeigt. Diese sind jedoch mit sehr großer Wahrscheinlichkeit in einem solchen Falle nicht fehlerhaft. Vielmehr liegt es an einer mangelnden Fehlersynchronisation des Compilers, der einmal durch einen Fehler „außer Tritt geraten" nicht weiter korrekt anaylsieren kann. Es ist nicht auszuschließen, dass bei komplexen Portierungsgegenständen solche Ursachen die Bottom-Up-Methode scheitern lassen.

(3) Adaption bezüglich der Rahmenbedingung Softwareumgebung

Diese Phase des Vorgehensmodells verursachte den größten Aufwand der Portierung. Konkret zeigte sich, dass besonders die Erstellung von separaten Softwaremodulen in Form von Gerätetreibern einen nicht unerheblichen Aufwand verursacht. Verallgemeinert auf ähnliche Portierungsprojekte, kann die Erstellung solcher zusätzlichen Softwaremodule im Rahmen der Adaption in das Modell aufgenommen werden. Unter der Voraussetzung, dass keine adäquaten Alternativen[1] zur Verfügung stehen, ist die Erstellung solcher Softwaremodule erforderlich. Denkbar wäre an dieser Stelle auch eine Art Portierungsbibliothek, welche die alten Bibliotheksfunktionen des Hostsystems kapselt, so dass eine Vielzahl von Altprogrammen ohne größeren Aufwand unter Verwendung dieser Bibliothek portiert werden können. Voraussetzung ist jedoch, dass keine speziellen Hardwarezugriffe verwendet werden.
Die im vorliegenden Fall erstellten Gerätetreiber erforderten ein enormes Maß an Einarbeitungezeit in die Materie der Treiberprogrammierung. Auch zeigte sich, dass die Treiberprogrammierung besonderes Fehlerpotenzial birgt, da mit einem fehlerhaften Treiber das komplette System destabilisiert werden kann. Sind diese Grundlagen allerdings erst einmal gelegt

[1] Im vorliegenden Fall gab es keine Alternativen, die den benötigten Funktionsumfang lieferten, Memory Mapped I/O fehlte immer.

und entsprechende Grundgerüste für Gerätetreiber entwickelt, fällt sicherlich nur noch ein Bruchteil des Aufwandes an.

(4) Test und Fehlerbereinigung

Der Schritt von Test und Fehlerbereinigung steht dem vorherigen Schritt in Sachen Aufwand an nichts nach. Er bildet die Grundlage für Aussagen darüber, ob das Portierungsziel erreicht wurde. Dieser Schritt besteht aus einem Zyklus von *Testen* und *Fehlerbereinigung*, bis das Portierungsziel erreicht oder schlimmstenfalls festgestellt wird, dass das bisherige Vorgehen in der Adaption falsch war.
Im vorliegenden Fall hat sich der diversifizierende Test in Form des Regressionstests[2] sehr bewährt. Ein anderes Vorgehen hätte wesentlich mehr Aufwand verursacht. Allein dadurch, dass Fehlverhalten mit Hilfe des Regressionstests eindeutig dem Portierungsgegenstand[3] oder der Portierung zugeordnet werden kann, wird gezielte Fehlerbereinigung ermöglicht.

Mit diesen Betrachtungen lässt sich das Vorgehensmodell etwas verfeinern (siehe Abbildung [8.1]):

[2] Abgesehen vom Test der Gerätetreiber, welche funktionsorientiert gegen ihre Spezifikation getestet wurden
[3] Damit ist die Ausgangsversion des Portierungsgegenstandes gemeint.

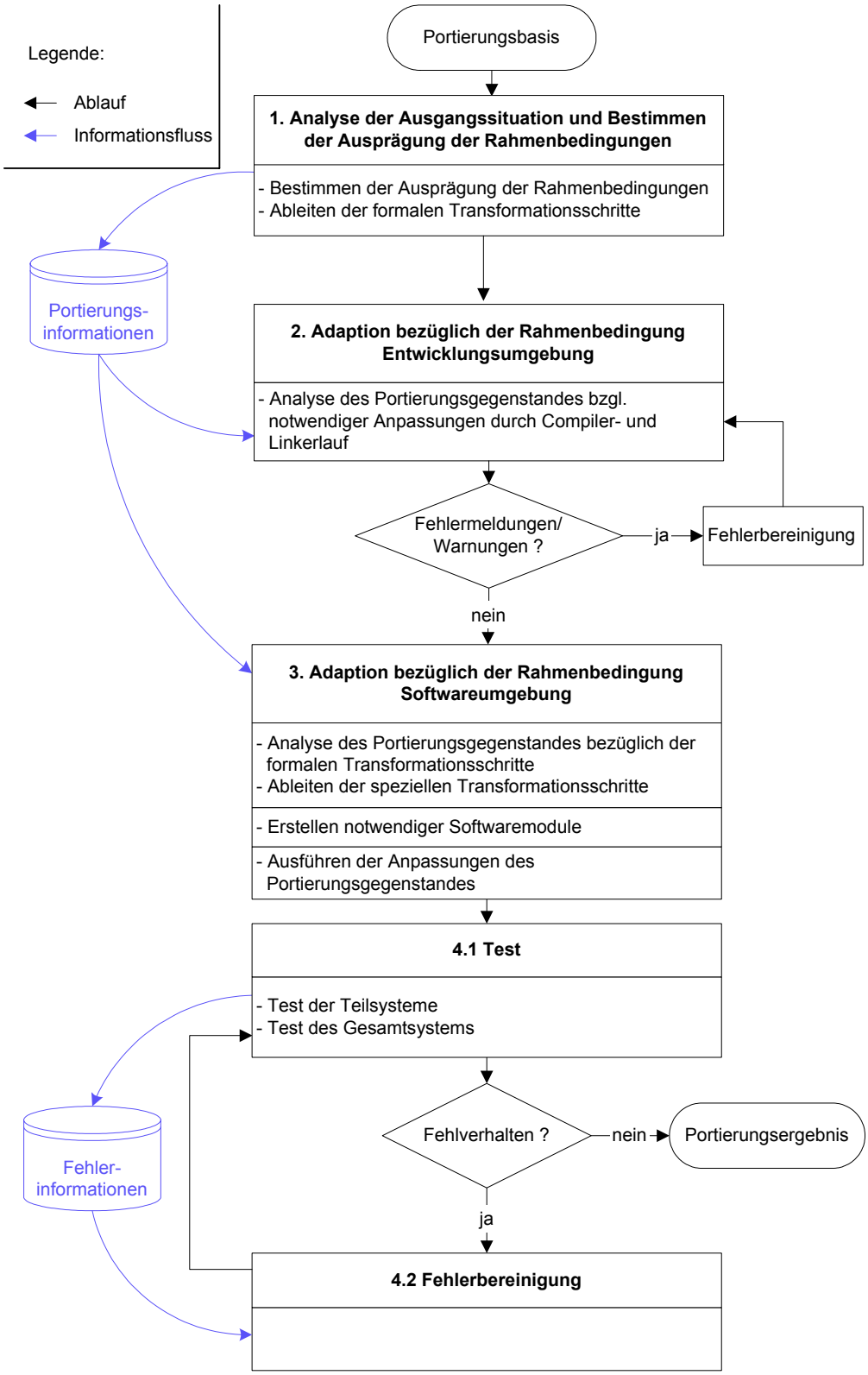


Abbildung [8.1] verfeinertes Vorgehensmodell

Die nachfolgende Tabelle [8.1] zeigt die Aufwendungen, die zur Durchführung der Portierung am XCTL-System von den Autoren aufzubringen waren. Dabei erheben diese Werte nicht den Anspruch allgemeingültig zu sein. Es hat sich gezeigt, dass der Erfahrungsschatz der Ausführenden einen wesentlichen Einfluss auf den Aufwand hat.
Die Tatsache, dass die Tests des portierten XCTL-Systems gezeigt haben, dass die Portierung erfolgreich war, bestätigt das entwickelte Vorgehensmodell. Ein Portierungsprojekt wie das vorliegende besitzt birgt ein nicht zu unterschätzendes Fehlerpotenzial und läuft Gefahr ohne entsprechende Koordinierung fehlzuschlagen. Demzufolge ist ein solches Projekt de facto nicht "ad hoc", also ohne systematisches Vorgehen zu portieren.

Aufwand für die Portierung von XCTL			
Schritt		**Aufwand in Tagen á 8h**	**Bemerkung**
1. Analyse der Ausgangssituation	Entwicklungsumgebung	ca. 1,5	
	Softwareumgebung	ca. 4	
	Hardwareumgebung	0	
2. Adaption - Entwicklungsumgebung	Analyse	ca. 12	
	Adaption		
3. Adaption - Softwareumgebung	Analyse	ca. 50	
	Adaption		
4. Test und Fehlerbereinigung	Test	ca. 30	
	Fehlerbereinigung	ca. 10	
Summe		117,5	Aufwand bei zwei Personen, entspricht 235 Mitarbeitertagen

Tabelle [8.1]

Auch wenn sich das entwickelte Vorgehensmodell am vorliegenden Projekt orientiert, so ist es auf ähnlich gelagerte Projekte sicher leicht zu übertragen. Der theoretische Teil dieser Arbeit diskutiert die nötigen Punkte und bietet damit zumindest einen allgemeingültigen Ansatz.

9. Ausblick

Nach der erfolgreichen Portierung des XCTL–Systems nach Windows 2000 hat sich an der Funktionalität des Systems nichts geändert. Die Architektur hingegen wurde den Gegebenheiten des neuen Betriebsystems angepasst. XCTL hat jedoch noch nicht das Stadium erreicht in dem man davon sprechen könnte, dass es ausgereift wäre. In diesem Abschnitt sollen nun Möglichkeiten der Weiterentwicklung des XCTL –Systems betrachtet werden.

Verbesserungen an der Architektur

Während der Portierung fiel bei der Auseinandersetzung mit dem Quelltext auf, dass die Architektur bestimmter Teile von XCTL einer Überarbeitung unterzogen werden sollte.

Motorkomponente

Dieser Teil von XCTL weist Mängel in der Umsetzung der Kommunikation mit der Hardware auf. Ein Großteil der Hardwarekommunikation wird mit Aufrufen der Wartefunktion *Delay()* realisiert (vergl. Kapitel Fehlerbeschreibung). Durch schlecht programmierte Warteschleifen können Aufrufe dieser Funktion, unnötiger Weise, derart aufsummiert werden, dass sich Wartezeiten ergeben, die so groß werden, dass bestimmte Timerereignisse nicht rechtzeitig behandelt werden können. Dieser Effekt tritt in Abhängigkeit der Größe von *dDelayMin* (siehe Kapitel Fehlerbeschreibung - Timingprobleme) auf (zu beobachten im Motoroptimierungsdialog). Da die ursprüngliche *Delay()* Funktion in ihrem Zeitverhalten in Abhängigkeit des Parameters nicht linear war, die neue *Delay()* Funktion dies aber ist, stellt sich die Frage, inwiefern die Werte, mit denen *Delay()* aufgerufen wird, nach der Portierung noch adäquat sind. Während der Arbeit an der Portierung fiel auf, dass die Motorhardware (besonders Typ C-812) sehr empfindlich auf Timingprobleme reagiert. Die Motorkomponente sollte deshalb unbedingt überarbeitet werden: Zum einen in Bezug auf die Wartezeiten, zum anderen in Bezug auf die unnötige Kaskadierung der *Delay()*-Aufrufe.
Ein weiterer Hinweis bezüglich der Motorkomponente betrifft den entsprechenden Teil in der Konfigurationsdatei *hardware.ini*. Es fiel

mehrfach auf, dass falsche Einträge den Einsatz von XCTL durch massives Fehlverhalten der Motorkomponente unmöglich machen, ohne dass auf die inkonsistenten Einträge der Konfigurationsdatei hingewiesen wird. Sicher ist ein gewisses Maß an Flexibilität in der Konfigurierbarkeit der Hardware nötig, aber dies darf nicht dazu führen, dass Einträge akzeptiert werden, die unsinnig sind.

Detektor Braun PSD

Die im Kapitel Fehlerbeschreibung erläuterten Fehler der Klasse *TBraunPSD* und der beschriebenen Fehler in den Messwerten lassen die Vermutung zu, dass sich noch einige Fehler dort befinden. Somit ist die Forderung nach dringender Überarbeitung diese Klasse gerechtfertigt.

Treiberarchitektur

Für die Kommunikation mit der Hardware wurde es nötig, die Treiberarchitektur von Windows 2000 in XCTL einfließen zu lassen. In der derzeitigen Version beschränkt sich die Funktionalität der einzelnen Gerätetreiber jedoch lediglich darauf, die E/A-Zugriffe an die Hardware weiter zu reichen[1]. Idealerweise könnte ein Großteil der Kommunikationsfunktionalität, die bislang in XCTL umgesetzt ist, in die Treiber verlagert werden. Ein erster Schritt in diese Richtung ist die Umsetzung des Gerätetreibers für den Detektor *Braun-PSD*. Durch eine solche Verlagerung von Code in den einzelnen Treiber würde zum einen die Schichtenarchitektur von XCTL perfektioniert, zum anderen kann damit auch der Quelltext von XCTL wesentlich vereinfacht werden. Beides Vorgänge, deren Umsetzung sich langfristig vorteilig sowohl auf Weiterentwicklung als auch auf Wartung von XCTL auswirken wird.
Ein weiterer Punkt die Treiberarchitektur betreffend, ergibt sich aus der Möglichkeit, die Simulationskomponenten von XCTL in spezielle Treiber auszulagern. Als Basis dafür können die am Rande der Gerätetreiberentwicklung entstandenen Testtreiber dienen. Sie bieten ein Interface, welches zu den echten Gerätetreibern identisch ist. Die Auslagerung des Codes hätte einige Vorteile. Neben den schon für die Gerätetreiber genannten käme als Vorteil hinzu, dass mit

[1] wobei bereits auf die Zulässigkeit einer Anforderung getestet wird, siehe Treiberspezifikation.

Simulationstreibern exakt der gleiche Code getestet würde, der auch im gleichen Anwendungsfall unter realen Bedingungen zum Einsatz kommt.

Multithreading

Im derzeitigen Zustand von XCTL fällt auf, dass das Programm in Lastsituationen nicht immer auf Benutzereingaben reagiert. Bestes Beispiel dafür ist die Startphase von XCTL. Wenn das XCTL-Hauptfenster kurz nach dem Start den Fokus verliert, ist es nicht möglich, zum Hauptfenster zurückzukehren, bis die Initialisierungsphase von XCTL beendet ist. Dies liegt daran, dass während dieser Phase keine Fensternachrichten behandelt werden, da der Prozeß mit E/A- oder sonstigen Funktionen beschäftigt ist. Um diese Probleme zu lösen, sollte *Multithreading* genutzt werden. Eine Zerschlagung des Einzelthreads in kleinere Threads würde eine Parallelisierung bewirken und somit Blockaden der Oberfläche verhindern helfen. Durch spezielle API-Bibliotheksfunktionen ist die Umsetzung von *Multithreading* einfach realisierbar.

Nutzen der Neuerungen von Windows 2000

Ein besonderes Merkmal von Windows 2000 ist die Verwaltung von Benutzerrechten. Durch dieses Feature ist man u.a. in der Lage, den Zugriff auf Teile eines Rechners für bestimmte Benutzer einzuschränken. Ein Konzept, welches besonders mit Blick auf die Verwendung der XCTL–Arbeitsplätze durch verschiedene Mitarbeiter und Studenten der Physik, die sich diese teilen müssen, von Bedeutung seien dürfte. Eine Anwendung dieses Konzepts in XCTL könnte zum Beispiel in Abhängigkeit vom gerade angemeldeten Benutzer eine entsprechende persönlich anpassbare Arbeitsumgebung für diesen zur Verfügung stellen.

Grafische Nutzerschnittstelle

Beim Umgang mit der Nutzerschnittstelle während Portierung und Test fiel auf, dass diese bestimmte ergonomische Unzulänglichkeiten aufweist. Als Beispiel seien hier unterschiedlich große “Abbruch“-“OK“-Buttonpaare, nicht einheitlich vertikal ausgerichtete Fensterelemente und die Informationsüberfrachtung bestimmter Dialoge (z.B. *Neue manuelle Justage* und *Protokollbuch*) genannt. Eine Überarbeitung der gesamten

Oberfläche mit der Umsetzung der Windows üblichen Optik wird in diesem Zusammenhang empfohlen. Die durchgeführte Anpassung der Statuszeile (siehe *Kapitel 7 Fehlerbereinigung, Statusze*ile) ist ein erster Schritt in diese Richtung.

Quellenverzeichnis

[ATOS 02] Hanisch, J. ; Letzel, J.
Anleitung zur Installation und zur Arbeit mit dem Tool ATOS
https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98y/werkzeuge/restricted/ATOS.htm
Berlin, 2002

[Axiom 92] AXIOM Technology Co., Ltd.
AX5216 5 Channel Counter/Timer Board – User's Manual
Taiwan, 1992

[Balzert 98] Balzert, H.
Lehrbuch der Softwaretechnik; Band 2
Spektrum Akademischer Verlag
Heidelberg, Berlin, 1998

[Borland 97] Borland Software Corpotation
Borland C++ 5.02 Online Help
1997

[Braun 94] M.Braun GmbH
Betriebsanleitung PSD ASA-System;
Schriftwechsel per Fax zwischen Herrn Leingartner (MBraun) und Herrn Damerow (Physik HUB) per Fax
Garching, 1994

[Bojic 02] Bojic, D.
Porting XCTL from Borland (16-bit) to Visual C++ 6.0 (32-bit)
[Projekt 98]
2002

[Buschhorn 88] Buschhorn, M ; Kuchnowski, H.J.
Entwicklung eines Portabilitätsmaßes und Entwurf eines Tools zur Unterstützung der Portierung, Diplomarbeit
Universität Dortmund
1988

[HaPaPi 00] Harder, R. ; Paschold, A. ; Picard, J.
Reverse-Engineering des Subsystems Detektoren des RTK-Steuerprogramms[1], *Studienarbeit*
Humboldt Universität zu Berlin
Berlin, 2000

[Jahnke 00] Jahnke, J. H. ; Walenstein, A.
Reverse Engineering Tools as Media for Imperfect Knowledge
Erschienen in: *7th Working Conference on Reverse Engineering*
IEEE Computer Society
LosAlamitos, Calif. 2000

[Kaindl 88] Kaindl, H.
Portability of Software
SIGPLAN Notices, Vol. 23, No. 6
1988

[Klösch 95] Klösch, R. ; Gall, H.
Objektorientiertes Reverse Engineering
Springer
Heidelberg, 1995

[Lauer 92] Lauer, T.
Programme für Windows 3.1 portabel anlegen
Erschienen in: *c't Magazin für Computertechnik*
Heise Zeitschriften Verlag
Hannover, 1992 Heft 8

[Liggesmeyer 02] Liggesmeyer, P.
Software-Qualität; Testen, Analysieren und Verifizieren von Software
Spektrum Akademischer Verlag GmbH
Heidelberg, Berlin 2002

[Messmer 98] Messmer, H.-P.
PC-Hardwarebuch; Aufbau, Funktionsweise, Programmierung 5. Auflage
Addison – Wesley Longman GmbH
Bonn, 1998

[1] RTK-Steuerprogramm ist die alte Bezeichnung des XCTL-Systems

[Microsoft 99a] Microsoft Corporation
Erste Schritte; Benutzerhandbuch zu Windows 2000
Microsoft Corporation
Ireland, 1999

[Microsoft 99b] Microsoft Corporation: *Microsoft Visual C++ 6 Programmierhandbuch; Der offizielle Leitfaden zur Programmierung mit Visual C++ 6*
Microsoft Press Deutschland
Unterschleißheim, 1999

[Mooney 97] Mooney, J.D.
Bringing Portability to the Software Process, Technical Report 97-1
Department of Statistics and Computer-Science, West Virgina University
Morgantown West Virgina
1997

[MSDN 03a] Microsoft Developer Network
MSDN Library
http://msdn.microsoft.com/library/default.asp
2003

[MSDN 03b] Microsoft Developer Network
MSDN DDK Library
2003
siehe Anhang C – Werkzeuge, DDK

[Müller 02] Müller, M.
Metriken zur Protabilitätsanalyse Windowsbasierter Software-Systeme; Diplomarbeit
Humboldt Universität zu Berlin
Berlin, 2002

[Oney 03] Oney, W.
Programming the Microsoft Windows Driver Model 2nd Edition
Microsoft Press (USA)
Redmond, Washington, 2003

[PI 95] Physik Instrumente GmbH
Information MS 40E, C-812 DC-Motor Controller
Operating Manual MS 37E, C-832 DC-Motor Controller
Waldbronn, 1993-1995

[Projekt 98] XCTL Projekt am Institut für Informatik der Humboldt Universität zu Berlin am Lehrstuhl für Softwaretechnik, Leitung Prof. Bothe
https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98y/index.html
Berlin, 1998-2003

[Radicon 95] Radicon Ltd., Scientific Instruments
General Description, Operators Manual – SCSCS
St. Petersburg, 1995

[Schützler 01] Schützler, K.
Wiedergewinnung von Subsystemen durch Use-Case-Analyse und Dateirestrukturierung am Beispiel des XCTL –Systems; Diplomarbeit
Humboldt Universität zu Berlin
Berlin, 2001

[Solomon 00] Solomon, D. A. ; Russinovich, M.
Inside Microsoft Windows 2000; Der offizielle Leitfaden zur Architektur der Betriebssystemplattform Windows 2000; 3. Auflage
Microsoft Press Deutschland
Unterschleißheim, 2000
http://www.sysinternals.com

Glossar

Adaptierung – spezifikationserhaltende Änderung eines Softwaresystems bzgl. der Funktion, Korrektheit und Effizienz.

API – Advanced Programming Interface, Funktionsbibliothek, die Systemfunktionen implementiert

Assembler – hardwarenahe Programmiersprache und gleichnamiges Werkzeug zum Übersetzen von Assemblerquelltext in Objektcode

Crossreferencer – Programm, welches die Aufrufbeziehungen der Funktionen eines im Quelltext vorliegenden Programms ermitteln kann

Decompiler – Werkzeug zur Rückgewinnung von Quellcode eines in einer höheren Programmiersprache geschriebenen, in binärer Form vorliegenden Programmes

Disassembler – Werkzeug zur Rückgewinnung von Assemblerquellcode aus einem in binärer Form vorliegenden Programmes

DLL – Dynamic Link Library, Programmbibliothek, die bei Bedarf (dynamisch) geladen werden kann.

DOS – Disk Operating System; einfaches textbasierendes Betriebssystem für PCs

GUID – Globally Unique Identifier, Ziffernfolge, die durch einen speziellen Algorithmus erzeugt wird, statistisch gesehen weltweit eindeutig, wird als Bezeicher für (weltweit) eindeutig zu identifizierende Elemente verwendet (z.B. eindeutige Bezeichnung für Interface eines Gerätetreibers)

Hostumgebung – Ausgangsumgebung des Softwaresystems (Umgebung, aus der das System portiert werden soll)

inline-Assembler – als inline-Assembler wird die Fähigkeit eines Hochsprachencompilers bezeichnet, Assemblerquelltext innerhalb eines Hochsprachenquelltextes verarbeiten zu können

IOCTL-Code – Input Output ConTroL Code, Kontrollcodes, die dazu dienen die Operationen von Gerätetreibern zu steuern.

IRP – Io Request Packet, Basisstruktur, die von E/A-Manager und Treibern genutzt wird um mit Treibern zu kommunizieren. Es existieren Haupt- und Neben-IRPs. Neben-IRPs spezifizieren die Anforderung die durch den Haupt-IRP gestellt wurde genauer.

Linker – bindet die Ausgaben eines Assemblers zu einem verwendbaren Endprodukt (z.B. ausführbares Programm)

Multitasking – gleichzeitige Ausführung von mehr als einem Programm

- kooperatives- : Variante des Multitasking bei der die Programme die Steuerung freiwillig an andere Prozesse abtreten
- preemptives- : Variante des Multitasking bei der ein Taskscheduler für die Vergabe der Prozessorzeit sorgt

Multithreading – gleichzeitiges Ausführen verschiedener Threads

PnP – Abkürzung für Plug and Play, Mechanismus, mit dessen Hilfe auf die manuelle Konfiguration von Erweiterungshardware verzichtet werden kann

Portierungsergebnis – das vollständig portierte Programm

Portierungsgegenstand – das zu portierende Programm

Profiler – Werkzeug, welches bestimmte Eigenschaften eines Programms zur Laufzeit ermitteln kann (z.B. Zeitverhalten)

Registry – Dieser englische Begriff bezeichnet im Zusammenhang mit Win32-Systemen die Systeminterne Datenbank zur Speicherung sämtlicher systembezogener Daten

Softwaresystem – Gesamtheit aller zu einem bestimmten System gehörenden einzelnen Softwarekomponenten

Systemumgebung – rechnergestütztes Informationssystem, bestehend aus Hardware, Systemsoftware und Anwendungssoftware (speziell Entwicklungssystem).

Task – Bezeichnung für ein Programm während seiner Ausführung

Taskscheduler – Softwarekomponente eines Betriebssystems, welche die Prozessorzeit des Systems nach bestimmten Prioritäten an die gleichzeitig laufenden Tasks vergibt

Thread – Programmteil, der parallel zu andern Programmteilen ausgeführt wird

Win16 – Kurzform für 16-Bit-Windows-Versionen (Windows 3.1x)

Win32 – Kurzform für 32-Bit-Windows-Versionen (Windows 9x/Me, Windows NT/2000/XP)

Zielumgebung – Zielumgebung des Softwaresystems (Umgebung, in der das System portiert werden soll)

Anhang A

Beschreibung der XCTL-Treiber

Der Funktionsumfang eines Treibers für XCTL lässt sich in zwei Kategorien einteilen:

1. Funktionen die vom Betriebssystem zur Verwaltung von Treiber und Geräten verlangt werden.
2. Funktionen, die den Datenaustausch mit der Hardware bewerkstelligen:
   -Grundfunktionalität: durch byte-weises Lesen oder Schreiben von Daten auf bzw. von der Hardware
   -erweiterte Funktionalität, durch Verlagerung von Funktionalität, die bislang in XCTL lokalisiert war, hinein in den Treiber

Um diese Funktionen und somit auch den gesamten Treiber Windows-konform zu erstellen, muss ein Gerätetreiber bestimmten Implementierungsrichtlinien entsprechen. Diese Richtlinien sind abhängig vom zu verwaltenden Gerät und von der Rolle, die der Gerätetreiber bei dieser Verwaltung einnimmt. Für die XCTL- Gerätetreiber gelten die Richtlinien für Funktionstreiber. Die Implementierungsrichtlinien werden an entsprechender Stelle in der folgenden Beschreibung der Gerätetreiber genannt.

Zuerst wird auf die in allen Treibern zu findenden Funktionen eingegangen. Danach werden die Unterschiede der einzelnen Treiber betrachtet.

1. Verwaltungsfunktionen

DriverEntry()

Ein Windowstreiber kann in gewisser Hinsicht vom Aufbau her mit einer DLL verglichen werden. Eine Sammlung von Funktionen wird wie in einer Bibliothek zur Verfügung gestellt, mit dem Unterschied, dass diese Funktionen nur für ein bestimmtes Gerät oder eine Geräteklasse dienen. Der zentrale Einstiegspunkt in einen Treiber wird durch die Funktion *DriverEntry()* gebildet. Diese Funktion ist die einzige, die in ihrem Namen festgelegt ist. Zwei Zeiger werden dieser Funktion übergeben: der erste

zeigt dabei auf das vom Betriebssystem angelegte Treiberobjekt und der zweite zeigt auf einen String, welcher den zum Treiber gehörenden Registrierungspfad enthält. In der Funktion müssen nun die weiteren Verwaltungsfunktionen (bzw. Dispatcherfunktionen) dem E/A Manager bekannt gemacht werden, da diese keine festdefinierten Namen haben. Innerhalb des übergebenen Treiberobjektes gibt es ein Feld von Zeigern und einige einzelne Zeiger, die nun mit den Adressen der Funktionen versehen werden müssen, für die die Zeiger stehen. Des Weiteren können in *DriverEntry()* treiberglobale Initialisierungen vorgenommen werden, was im Falle der XCTL-Treiber jedoch nicht nötig war.
Nun zur Beschreibung der Zeigerinitialisierungen.
Dem Pointer *DriverUnload* muss in jedem Falle die Adresse einer Funktion zugewiesen werden, welche all das rückgängig macht (allokierten Speicher freigeben etc.), was in *DriverEntry()* angefordert bzw. eingerichtet wurde. In den XCTL-Treibern heißt diese Funktion *XxUnload()*[1]. Diese parameterlose Funktion ist leer, da in *DirverEntry()* nichts durchgeführt wurde, was hier rückgängig gemacht werden müsste.
Für die Behandlung der Haupt-IRPs existiert im Treiberobjekt ein Zeigerfeld, welches die Adressen der Dispatcherfunktionen für die jeweiligen IRPs aufnehmen soll. In den XCTL-Treibern werden die Haupt IRPs *IRP_MJ_CREATE* und *IRP_MJ_CLOSE* mit der Adresse der Funktion *XxDispatchCreateClose()* initialisiert. Diese IRPs werden vom E/A -Manager genau dann an den Treiber eines Gerätes geschickt, wenn ein Handle auf einen Gerätetreiber angelegt werden soll (via *CreateFile()*) bzw. wenn ein Handle geschlossen werden soll (via *CloseHandel()*). Da die XCTL- Treiber in ihrer aktuellen Funktionalität keine Aktionen zu diesen Zeitpunkten ausführen müssen, soll für beide Fälle ein und dieselbe Funktion aufgerufen werden (*XxDispatchCreateClose()*), die im Wesentlichen nichts weiter tut, als die IRP-Anfrage zu vervollständigen.
Dem Haupt IRP IRP_MJ_DEVICE_CONTROL wird die Adresse der Funktion *XxDispatchDeviceControl()* zugeordnet. Diese wird vom E/A Manager immer dann aufgerufen, wenn eine Funktion via *DeviceIoControl()* aus einer Benutzeranwendung angefordert wird. Innerhalb dieser Funktion werden dann entsprechend des IOCTL-Codes die angeforderten Funktionen aufgerufen. Der Rückkehrcode einer so angeforderten Funktion wird dann ausgewertet und der IRP damit vervollständigt, so dass das Programm, welches die IOCTL Anforderung

[1] Erklärung für X: An dieser Stelle sei daraufhin gewiesen, dass Funktionen in den Motortreibern mit *Mc* für MotorControl beginnen, Funktionen für Detektortreiber mit *Dc* für DetectorControl

initiiert hat, auch über den Erfolg der Treiberoperation in Kenntnis gesetzt ist.
Der letzte der Haupt-IRPs, die von den XCTL-Treibern behandelt werden, ist der IRP *IRP_MJ_PNP* . Dieser wird vom PnP-Manager verschickt, um die vom Treiber verwalteten Geräte zu steuern. Zu diesen Steueraufgaben zählen beispielsweise das Starten und Stoppen eines Gerätes. Aber auch Energiefunktionen wie der Stand-by Modus werden dem Gerät (also der Treiberinstanz, die für ein bestimmtes Gerät zuständig ist) mittels *IRP_MJ_PNP* mitgeteilt. *XxDispatchPNP()* heißt die Funktion, die in den Treibern verwendet wird, um dieses Major IRP zu behandeln. Innerhalb von *XxDispatchPNP()* werden dann die Neben-IRPs bearbeitet.
Als letztes wird in *DriverEntry()* die Adresse der AddDevice Funktion festgelegt auf die Adresse der Funktion *XxAddDevice()*. Die Implementation dieser Funktion wird für PnP-Geräte verlangt, um neue Geräte zum System hinzufügen zu können. Obwohl es sich bei der von XCTL verwendeten Hardware nicht um PnP-Geräte, handelt muss diese Funktion implementiert sein, um die Verwaltung mehrerer Geräte des gleichen Typs über ein und denselben Gerätetreiber ermöglichen zu können, dazu jedoch mehr in der Erklärung der *XxAddDevice()* Funktion. Damit ist die *DriverEntry()* Funktion vollständig.

XxAddDevice()

Die Funktion *XxAddDevice()* hat die Aufgabe, ein neues Gerät dem System hinzuzufügen, der Art, dass sämtlich nötige Initialisierungen der Treiberstrukturen initialisiert werden. Immer dann, wenn der PnP-Manager erkannt hat, dass ein Gerät eines bestimmten Typs neu an das System angeschlossen wurde, wird diese *AddDevice*-Funktion aufgerufen. Dabei kann der passende Treiber entweder mit einer oder mehreren Instanzen bereits geladen worden sein, oder aber er wird automatisch geladen. Bei PnP –Geräten wird die Präsenz neuer Hardware vom PnP-Manager automatisch erkannt, bei nicht PnP–Geräten muss dem System die Anwesenheit neuer Hardware über den Hardwarewizard (vergl. Anhang – C Werkzeuge) mitgeteilt werden. Ebenso wird die *AddDevice*-Funktion beim Systemstart aufgerufen, wenn die Hardware dem System bereits bekannt ist, also der Gerätetreiber bereits erfolgreich installiert wurde. Je nach dem um was für einen Typ Gerätetreiber es sich handelt, hat eine *AddDevice* –Funktion bestimmte Aufgaben zu erledigen. Im Falle der XCTL-Gerätetreiber vom Typ Funktionstreiber sind dies folgende:

1. Erzeugen des Geräteobjektes
2. Initialisierung der Geräteerweiterung (DeviceExtensions)
3. Registrierung des Geräteinterfaces
4. Anhängen des Gerätes an den Gerätestack

Diese Aufgaben nun im einzelnen genauer.
Die erste Aufgabe einer *AddDevice*-Funktion ist die Erzeugung eines Geräteobjekts[2]. Dieses Geräteobjekt kann wahlweise einen Namen erhalten oder, wie im Falle der XCTL Gerätetreiber, unbenannt bleiben. Dies geschieht durch den Aufruf der Funktion *IoCreateDevice()*, damit werden der nötige Speicher für ein solches Objekt allokiert und einige Felder des Geräteobjektes initialisiert. Wesentlich bei diesem Schritt ist, dass die sog. Geräteerweiterung berücksichtigt wird. Diese kann als geräteobjekteigener Speicher verstanden werden, der im weiteren Verlauf dazu dient, objektbezogene Daten aufzunehmen, die je nach Treibertyp unterschiedlich aufgebaut sein können. Im Falle der XCTL-Treiber ist diese Geräteerweiterung eine selbstdefinierte Struktur, welche auf den entsprechenden Treiber ausgelegt ist. *IoCreateDevice()* wird der für die Geräteerweiterung benötigte Speicherbedarf übergeben und nach dem erfolgreichen Erzeugen des Geräteobjekts ist die Geräteerweiterung über einen Zeiger im Geräteobjekt verfügbar. Die Initialisierung der Geräteerweiterung ist sodann auch die nächste Aufgabe der *AddDevice*-Funktion, für Einzelheiten, welche Werte in der Geräteerweiterung gespeichert werden, sei an dieser Stelle auf die entsprechende, kommentierte Quelltextpassage verwiesen.
Da die Autoren sich bei der Entwicklung der Treiber dazu entschieden haben, die Geräte über registrierte Interfaces und da das Geräteobjekt bei seiner Erzeugung auch keinen Namen erhalten hat, muss nun in der *AddDevice*-Funktion das Interface für das soeben erzeugte Geräteobjekt registriert werden. Dies geschieht mit Hilfe der Funktion *IoRegisterDeviceInterface()*. Wichtigster Parameter dieser Funktion ist die *InterfaceClassGuid*. Dabei handelt es sich um eine GUID, welche die Geräteinterfaceklasse weltweit eindeutig identifiziert [Oney 03].
IoRegisterDeviceInterface() konstruiert, unter Zuhilfenahme dieser GUID, eines Instanzzählers und weiterer Informationen einen eindeutigen Interfacenamen, über den sich dann das aktuell in der Initialisierung befindliche Gerät ansprechen lässt. Dieses Verfahren hat gegenüber der Benennung des Gerätes bei der Erzeugung via *IoCreateDevice()* mit einem statischem String den Vorteil, dass sich der Entwickler zum einen nicht

[2] Softwarerepräsentation eines konkreten Hardwaregerätes

um mehrere Geräteinstanzen kümmern muss und zum anderen, dass ein registriertes Geräteinterface auch gegen unberechtigte Benutzung geschützt werden kann.
Als nächstes ist in der *AddDevice*-Funktion das aktuell erzeugte Gerät an den Gerätestack anzuhängen. Dabei wird die Funktion *IoAttachDeviceToDeviceStack()* aufgerufen. Durch diesen Schritt wird sichergestellt, dass die Nachrichtenübermittlung via IRPs an die Treiber im Gerätestack lückenlos funktioniert.
Die letzte Aufgabe der *AddDevice*-Funktion besteht darin, einige Geräteflags nach Bedarf zu setzen und den Zeiger auf das unterliegende Geräteobjekt zu speichern, damit beim Entfernen des aktuell eingerichteten Gerätes kein Loch in die Treiberkette gerissen wird.

XxDispatchPNP()

Diese Funktion, ebenfalls aus der Kategorie der Verwaltungsfunktionen, übernimmt die Bedienung der an die Treiberinstanz vom E/A Manager gesendeten IRPs. Zuerst wird, wie in sämtlichen Funktionen der XCTL - Treiber, die Geräteerweiterung innerhalb der Funktion verfügbar gemacht, durch die das gegenwärtig zu bedienende Gerät festgelegt ist.
An dieser Stelle wird das Prinzip der „Quasi-“ Objektorientierung in der Treiberprogrammierung sichtbar, da jede Funktion einen Zeiger auf das zu verwendende Geräteobjekt bekommt[3]. Man könnte diesen in gewissem Sinne mit dem versteckten *this* –Zeiger vergleichen, den jede Memberfunktion einer Klasse in C++ mit übergeben bekommt. Die Identifizierung des aktuellen Objektes ist also gewährleistet, jedoch nicht wie in C++ automatisch, da diese Verantwortung beim Treiberentwickler liegt, der diesen Geräteobjektzeiger korrekt einzusetzen hat. Im Falle der XCTL- Treiber dient dieser Zeiger u.a. dazu, die Geräteerweiterung des entsprechenden Gerätes in der jeweiligen Funktion zu Verfügung zu stellen.
In der MSDN Hilfe des Windows DDK [MSDN 03b] wird eine Liste angeführt, welche die IRPs enthält, die für einen WDM Funktionstreiber nötig sind. Ausgehend von der Tatsache, dass die XCTL-Treiber keine PnP -Hardware zu steuern haben, das Verhalten der Treiber jedoch trotzdem einige PnP Eigenschaften umfasst (z.B. mehrere Geräte über einen Gerätetreiber steuern zu können), wurden bestimmte IRPs ausgewählt, die

[3] Anmerkung: mit freierhältlichen Tools (DDK) ist nur normales C möglich, kein C++

von der *DispatchPnP*-Funktion bedient werden. Folgende IRPs werden unterstützt und auch von [MSDN 03b] gefordert:

IRP_MN_START_DEVICE
IRP_MN_QUERY_REMOVE_DEVICE
IRP_MN_REMOVE_DEVICE
IRP_MN_CANCEL_REMOVE_DEVICE

Die folgenden vier IRPs werden zwar auch gefordert, werden aber durch die *DispatchPnP*-Funktionen der XCTL-Treiber nicht unterstützt:

IRP_MN_QUERY_STOP_DEVICE
IRP_MN_STOP_DEVICE
IRP_MN_CANCEL_STOP_DEVICE
IRP_MN_SURPRISE_REMOVAL

Dies wurde deshalb so entschieden, da diese IRPs im Zusammenhang mit Geräteeigenschaften stehen, die ausschließlich von PnP-tauglichen Geräten unterstützt werden, die alte XCTL-Hardware jedoch, wie bereits dargelegt, nicht zur Gruppe der PnP-tauglichen Geräten zählt. IRPs mit der Bezeichnung IRP_MN_*_STOP_DEVICE dienen laut MSDN DDK Beschreibung der Umorganisierung der Systemressourcen durch Windows. Eine solche Situation kann jedoch keinen Einfluss auf die XCTL-Hardware haben, da die Änderung deren Ressourcen einer manuellen Umschaltung der Jumper und Dip-Switches auf den Karten bedarf. Die benötigten Ressourcen und die Tatsache, dass es sich dabei um ausschließlich manuell einstellbare Ressourcen handelt, werden dem System (der Komponente PnP-Manager) über eine Einstellung in der Installationsdatei der Treiber mitgeteilt (siehe folgenden Abschnitt *Installationsumgebung*), so dass der PnP-Manager nie die Anforderung auf Umordnung der Ressourcen an einen XCTL-Treiber senden wird.
IRP_MN_SURPRISE_REMOVAL wird vom PnP-Manager verschickt, wenn das zugehörige Gerät ohne das Wissen des Systems (Komponente PnP-Manager) aus dem System entfernt wird, was jedoch bei den XCTL-Karten deshalb nicht möglich ist, da sie im Computer fest eingebaut sind, demzufolge muss auch dieses IRP nicht unterstützt werden.
Doch nun zu den von den XCTL-Treibern aktuell unterstützten IRPs im einzelnen:

IRP_MN_START_DEVICE

Dieses IRP wird verschickt, wenn der PnP-Manager für das entsprechende Gerät die Ressourcen zugewiesen hat. Wie oben erwähnt, handelt es sich bei dieser Zuweisung um keine durch den PnP-Manager ausgeführte Zuweisung, sondern es werden lediglich die bei der Geräteinstallation angegebenen Ressourcen übergeben. Die Eingabeparameterliste dieses IRP enthält eine Liste mit eben diesen Ressourcen, so dass der Treiber für das entsprechende Gerät (identifiziert über den Geräteobjektzeiger) nun die Ressourcen erfahren kann.

Bei Eintreffen dieses IRP wird im Wesentlichen folgendes getan:

Die übermittelten Ressourcen werden dahingehend geprüft, ob sie den Anforderungen des jeweiligen Gerätes entsprechen.

Wenn die Ressourcen gültig sind, werden sie in der Geräteerweiterung des aktuellen Gerätes abgespeichert, um diese somit für die weitere Verwendung durch andere Funktionen zur Verfügung zu stellen.

Als nächstes findet die Prüfung der Hardware mittels der Funktion *IOCheckCard()* statt, um zu testen, ob sich überhaupt entsprechende Hardware im PC befindet. Diese Funktion ist in jedem Treiber anders aufgebaut, da die Erkennung bestimmter Hardware sehr spezifisch ist.

Da im Normalfall das Gerät nun einsatzbereit ist, muss es nun so im System zur Verfügung gestellt werden, dass andere Software darauf zugreifen kann. In der AddDevice- Funktion wurde bereits die entsprechende Geräteinterfaceklasse registriert, so dass jetzt lediglich eine Instanz dieser Geräteinterfaceklasse aktiviert werden muss. Dies geschieht durch Aufruf der Funktion *IoSetDeviceInterfaceState()* mit dem entsprechenden Gerätenamen, der in der *AddDevice*-Funktion erzeugt und in der Geräteerweiterung abgespeichert wurde. Jedes Ergebnis dieser Operationen wird in einem entsprechenden Feld der Geräteerweiterung gespeichert, so dass beim Eintreten in den Zweig IRP_MN_REMOVE_DEVICE auch nur das rückgängig gemacht wird, was rückgängig gemacht werden kann, dazu jedoch mehr im Abschnitt zu IRP_MN_REMOVE_DEVICE.

Sollte eine der beschriebenen Operationen aus irgend einem Grunde fehlschlagen, so bricht *XxDispatchPNP()* mit dem entsprechenden Fehlercode ab und übergibt die Ausführung an den aufrufenden Prozess zurück, der darauf hin den IRP zum Entfernen des Geräts an den Treiber sendet. Das Starten des Gerätes schlug somit fehl. Dem Benutzer wird dies durch ein gelbes Ausrufezeichen im Gerätemanager am Symbol des entsprechenden Gerätes angezeigt.

IRP_MN_QUERY_REMOVE_DEVICE

Wenn der PnP-Manager den Treiber anweist, ein bestimmtes Gerät zu entfernen, wird dem Treiber dieses IRP übermittelt. In den XCTL-Treibern wird dieses IRP dazu genutzt, den Gerätestatus so zu ändern, dass bei Eintreffen eines IRP_MN_REMOVE_DEVICE entschieden werden kann, ob das Gerät tatsächlich entfernt werden soll, da der Ablauf für die Anforderung, ein Gerät zu entfernen, immer diese ist: erst trifft IRP_MN_QUERY_REMOVE_DEVICE ein, dann IRP_MN_REMOVE_DEVICE

IRP_MN_REMOVE_DEVICE

Wenn dieses IRP empfangen wird, soll das Gerät entfernt werden. Wichtig dabei sind jedoch die Vorbedingungen, unter denen dieses IRP verschickt wird. Zum einen kann diesem IRP ein IRP_MN_QUERY_REMOVE_DEVICE vorausgehen, was ein geplantes Vorgehen beim Entfernen des Gerätes anzeigt. Zum anderen können ungeplante Vorgänge dazu führen, dass IRP_MN_QUERY_REMOVE_DEVICE empfangen wird. Dies ist zum Beispiel dann der Fall, wenn der Start des Gerätes unvorhersehbar fehl schlug oder wenn das Gerät überraschend entfernt wurde, was bei der XCTL-Hardware jedoch, wie bereits erwähnt, nicht eintreten kann.

In den XCTL-Treibern wird nun bei Eintreffen dieses IRPs überprüft, inwiefern es sich um eine gültige Anforderung handelt (also entweder ein IRP_MN_QUERY_REMOVE_DEVICE voran ging, oder aber der Gerätestart fehlschlug). Abhängig davon, welche Operationen beim Gerätestart auch tatsächlich erfolgreich waren, werden diese in umgekehrter Reihenfolge dann rückgängig gemacht (Flags im Feld *DevInitState* der Geräteerweiterung).

IRP_MN_CANCEL_REMOVE_DEVICE

Mit dem Empfang dieses IRP wird die Anforderung nach Entfernen des Gerätes durch das System wieder aufgehoben. Entsprechend dazu wird in den XCTL-Treibern an dieser Stelle der Gerätestatus in der Geräteerweiterung angepasst, also der Status zum Entfernen des Gerätes aufgehoben.

Trifft ein nicht unterstütztes IRP ein, so wird im *default* -Zweig der IRP nicht angetastet und an den nächstunterliegenden Treiber weitergereicht.
Konnten die implementieren IRPs erfolgreich bearbeitet werden, wird der entsprechende Stauswert gesetzt und die IRP-Anforderung weitergeleitet.

XxDispatchDeviceControl()

Dies ist die letzte der Verwaltungsfunktionen. Sie wird aufgerufen, wenn von einem Programm ein Zugriffsversuch auf den Treiber ausgeht. Hauptaufgabe dieser Funktion ist die Bedienung der IOCTLs, die von einem solchen Pogramm an den Treiber geschickt werden. In einer *switch()* –Anweisung werden sämtliche vom Treiber unterstützten IOCTLs bedient. Welche IOCTLs ein bestimmter XCTL Treiber unterstützt kann, in der zu ihm gehörenden Datei „*IOCTL.h“ ermittelt werden.
Es existieren mindestens drei Gruppen von IOCTLs: zum Lesen und Schreiben von Ports und zum Ermitteln der ID[4] der Treiberinstanz. Die Funktionen *XxIoctlReadPort()* und *XxIoctlWritePort()* werden entsprechend den IOCTLs IOCTL_Xx_READ_* bzw. IOCTL_Xx_WRITE_* aufgerufen. Bei Eintreffen des IOCTL-Codes IOCTL_Xx_REPORT_ID wird die in der Geräteerweiterung gespeicherte ID (also die Basisadresse bzw. der Basisport) an den Sender des IOCTL-Codes übermittelt.
Zuletzt vervollständigt *XxDispatchDeviceControl()* den IRP-Request.

XxIoctlReadPort() / XxIoctlWritePort()

Diese Funktionen führen die Lese/Schreibzugriffe auf die Hardware durch. Angelegt sind sie (in Abhängigkeit vom IOCTL) auf BYTE-, WORD- und DWORD- Zugriffe, momentan nutzt XCTL jedoch nur den BYTE-weisen Zugriff auf die Hardware.
Im Wesentlichen werden in diesen Funktionen die benötigten Kommunikationspuffer überprüft, die zuzugreifenden Adressen werden auf Zulässigkeit überprüft und die Hardwarezugriffe ausgeführt. Dadurch dass die übermittelten Adressen überprüft werden, kann in einem gewissen Rahmen, sichergestellt werden, dass ein XCTL-Treiber nicht als generischer Port-I/O-Treiber missbraucht wird bzw. keine Ressourcen angetastet werden, die nicht zur Verfügung gestellt wurden.

[4] Die Bezeichnung ID, für Identifier, wurde gewählt, da die Controller unterschiden werden müssen. Dafür bot sich die Basisadresse der Karte an. Da jedoch die Basisadresse vom Betriebssystem verwaltet wird, XCTL mit dieser also keine Zugriffe selbst ausführt, wurde ID als Bezeichner gewählt. Über ihn wird der passende Treiber selektiert, und nicht, wie die Bezeichnung Basisadresse vermuten lassen könnte, dem Treiber die Basisadresse für seine Zugriffe zugewiesen.

IOCheckCard()

Diese Funktion wurde in allen Treibern (außer den Testtreibern) implementiert und dient der Erkennung der anzusteuernden Hardware. Demzufolge unterscheiden sich alle Treiber im Code dieser Funktion, da er für jede Controllerkarte speziell ist. Größtenteils wurde der Code aus bestimmten Methoden der entsprechenden Hardwareklassen von XCTL entnommen, welche dies genau sind, kann der folgenden Tabelle entnommen werden. Wenn die Hardware ordnungsgemäß erkannt wurde, liefert die Funktion einen Statuscode zurück der den Erfolg der Hardwareerkennung anzeigt (STATUS_SUCCESS), andernfalls wird das Fehlschlagen angezeigt (STATUS_UNSUCCESSFUL).

Herkunft der Codefragmente für die Hardwareerkennung in *IOCheckCard()*	
XCTL-Treiber	**Ursprung Codefragment**
MC812	TC_812ISA::CheckBoardOk()
MC832	TC_832::CheckBoardOk()
DCGeneric	TGenericController::SelectChip()
DCRadicon	abgeleitet aus Dokumentation
DCBraunPSD	abgeleitet aus Dokumentation

Unterschiede der XCTL-Gerätetreiber

Die Treiber unterscheiden sich jedoch auch an einigen Stellen. Die Unterschiede werden im Folgenden behandelt:

DCBraunPSD

Der Gerätetreiber für den BraunPSD nimmt eine gewisse Sonderstellung ein, da dieser in besonderer Hinsicht auf die Ablösung der alten 16-bit ASA.Dll entwickelt wurde, dadurch weicht die Liste der unterstützten IOCTLs erheblich von denen der anderen XCTL - Treiber ab. Folgende IOCTLs werden von diesem Treiber unterstützt:

IOCTL_DC_REPORT_ID
IOCTL_DC_READ_BYTE
IOCTL_DC_WRITE_BYTE
IOCTL_DC_SET_WFR_CYCLES
IOCTL_DC_GET_DATA

Bei den ersten drei IOCTLs verhält sich der BraunPSD Treiber identisch, wie alle anderen XCTL-Treiber auch. IOCTL_DC_SET_WFR_CYCLES ersetzt die *SetTimeOut()* Funktion der ASA.DLL, welche einen schleifenbasierten Timer mit der übergebenen Zyklenanzahl initialisiert. IOCTL_DC_GET_DATA implementiert die *GetData()* Funktion der ASA.DLL. Sie dient dem Auslesen einer bestimmten Anzahl von Messkanälen. Die Struktur *IrpParamsGetData* wird dabei mit einem Zeiger und der Anzahl der zu lesenden Worte gefüllt und an *DeviceIoControl()* übergeben. Der Zeiger muss auf den Speicherbereich zeigen, der die Messkanäle aufnehmen soll, die Allokation dieses Speichers ist von der aufrufenden Applikation zu erledigen. Der Rückkehrwert dieses IOCTLs gibt die Anzahl der tatsächlich gelesenen WORDs an. [Braun 94]

DCGeneric

Dieser Treiber unterscheidet sich nur in einem Punkt von den anderen. Er besitzt zur Vereinfachung des Hardwarezugriffs in der Funktion *IOCheckCard()* die Funktionen *ReadPort()* und *WritePort()*. Diese verwenden wie *XxIoctlReadPort()* bzw. *XxIoctlWritePort()* auch die Makros *READ_PORT_UCHAR()* bzw. *WRITE_PORT_UCHAR()*, die vom DDK bereitgestellt werden, jedoch wird zusätzlich noch eine definierte Zeitspanne nach dem Hardwarezugriff gewartet, um der Karte Zeit zu geben, die Anforderungen zu verarbeiten.

MC812

Dieser Motorcontrollertreiber weicht in seiner Implementation dahingehend von den anderen XCTL-Treibern ab, dass die Hardwarezugriffe nicht über Port I/O abgewickelt werden, sondern via Memory Mapped I/O. Dieser Unterschied ist an einigen wenigen Implementationsunterschieden zu erkennen: die Funktionen *McMapIo()* bzw. *McUnMapIo()* sind nur in diesem Treiber vorhanden, für den Zugriff auf die Hardware werden in diesem Treiber die DDK-Makros READ_REGISTER_* bzw. WRITE_REGISTER_*[5] verwendet. Dies resultiert aus der Tatsache, dass der Hardwarezugriff via Memory Mapped I/O unter Win32 zuvor der Verwendung eines Mappings des betroffenen Speicherbereiches bedarf. Dieses Mapping wird von der

[5] * steht dabei für UCHAR, WORD, DWORD

Funktion *McMapIo()* erledigt. Dabei wird mit Hilfe der Bibliotheksfunktion *MmMapIoSpace()* der zu verwendende physikalische Adressbereich in den Systemspeicher eingeblendet, nur dann kann mittels der beschriebenen Makros auf diesen Bereich zugegriffen werden.

Verwendung der Treiber

Im Anhang B *Installationsanleitungen* finden sich ausführliche Installationsanleitungen für alle Treiber. Für den Einsatz der Treiber ist es wichtig, ob eine Checked- oder Free-Build Variante verwendet wird. Im Windows DDK wird zwischen diesen beiden Varianten unterschieden, um dem Entwickler bessere Möglichkeiten der Fehlersuche bei der Treiberentwicklung zu geben. Checked Build bedeutet dabei Debug-Version, also eine Version, die unter den Gesichtspunkten des Debuggings übersetzt wurde. Free Build hingegen bedeutet Release Version, also eine Version, bereinigt von allen Symbolen und Testausgaben. Ausgehend davon sollte auch der Einsatz der entspechenden Varianten überprüft werden. Für die Entwicklung der Hardwaresteuerung in XCTL empfehlen sich die Testtreiber und die echten Hardwaretreiber als Checked Build Variante. Bei diesen wird bei jeder ausgeführten Operation eine Zeichenkette an den Kerneldebugger (siehe Anhang C- Werkzeuge) abgeschickt. Das bedeutet, dass mit Hilfe eines Kerneldebuggers während der Ausführung von XCTL die Kommunikation zwischen Applikation, Treiber und Karte in einem bestimmten Rahmen überwacht werden kann. Alternativ kann auch z.B. das Hilfsprogramm DebugView verwendet werden (siehe Anhang – C *Werkzeuge*). Diese Ausgaben dieser Zeichenketten kosten jedoch ein gewisses Maß an Systemressourcen und Zeit, weshalb dringend dazu geraten wird, auf Systemen, die nicht der XCTL-Entwicklung dienen, ausschließlich Free-Build-Treibervarianten einzusetzen. Free-Build-Treiber haben aufgrund der fehlenden Ausgabemöglichkeit auch einen wesentlich kleineren Binärcode.

Im Abschnitt *CVS-Modul* weiter unten ist beschrieben, wo sich die jeweiligen Varianten im CVS befinden.

Testtreiber

Motivation

Im Verlauf der Portierung und der Treiberentwicklung stellten die Autoren fest, dass es eine Hilfe wäre, einen Treiber verwenden zu können, der die Kommunikation zwischen Applikation und Treiber in einer Log-Datei festhält. Eine Implementation der umfangreichen Registry- und der Dateiverwaltungsfunktionen in die eigentlichen XCTL-Treiber hätte eine potentielle Fehlerquelle mehr für den Code dargestellt und die Übersichtlichkeit des Quelltextes verschlechtert, dazu kommt auch noch, dass diese Funktionalität nur zur Entwicklungszeit benötigt wird. Aus diesen Beweggründen entstanden die XCTL-Testtreiber. Es handelt sich dabei um einen Satz Treiber, welcher für jeden echten Hardware XCTL-Treiber ein entsprechendes Testpendant zur Verfügung stellt, der sich von dem Hardwaretreiber in einigen Punkten wesentlich unterscheidet.

Umsetzung

Wie bereits erwähnt, kommen diese Treiber ohne jegliche XCTL-Hardware aus, stellen aber nach außen das gleiche IOCTL-Interface zur Verfügung wie die echten Hardwaretreiber. Wichtig dabei ist zu erwähnen, dass die Testtreiber in ihrem aktuellen Zustand nicht in der Lage sind entsprechende Hardware zu simulieren. Ihre Funktionalität beschränkt sich darauf, ein korrektes Treiberinterface anzubieten und sämtliche Anforderungen an den Treiber sowie Fehlermeldungen in einer Log-Datei im Systemverzeichnis mitzuprotokollieren. Wenn durch die den Treiber verwendende Applikation eine Leseanforderung gestellt wird (zB: IOCTL_MC_READ_BYTE), dann wird ein Wert zurückgeliefert ohne jeglichen Sinngehalt. Einzig der IOCTL-Code IOCTL_MC_REPORT_ID wird identisch wie bei einem echten XCTL-Hardwaretreibers behandelt.

Verwendung

Die Testtreiber können dafür eingesetzt werden, die Kommunikation zwischen Applikation und Treiber ausgiebig zu testen. Ebenso können sie, in Grenzen, dazu dienen, die von der Applikation ausgehenden Befehlsströme an die Hardware aufzuzeichnen. Zur Verwendung dieser Treiber wurden im Modul HWIO.h zwei Direktiven vorgesehen, mit Hilfe derer zwischen echten Hardwaretreibern und Testtreibern umgeschaltet werden kann: DETECTORS_TESTDRIVER und MOTORS_TESTDRIVER. Die Installation der Testtreiber läuft nicht wie bei den echten Hardwaretreibern ab, dazu sei jedoch auf die Installationsanweisung verwiesen, Anhang B.
Jeder Testtreiber erzeugt ab dem Zeitpunkt seiner erfolgreichen Installation eine Log-Datei im Systemverzeichnis[6]. Der folgenden Tabelle können die Log-Dateinamen der einzelnen Testtreiber entnommen werden.

Log-Dateinamen der XCTL-Testtreiber	
Testtreiber	**Log-Dateiname**
MC812Test	MC812Test.log
MC832Test	MC832Test.log
DCGenericTest	DCGTest.log
DCRadiconTest	DCRTest.log
DCBraunPSDTest	DCBTest.log

Tabelle [A.1]

Außerdem existieren die Testtreiber ausschließlich als Checked-Build, d.h., sie senden immer Debuggingstrings, die ebenfalls dazu dienen können, die Kommunikation Treiber - Applikation zu überwachen.

Ausblick

Die Testtreiber sind in ihrem derzeitigen Zustand durchaus darauf ausgelegt, zu Simulationstreibern weiterentwickelt zu werden. Eine Möglichkeit bestünde zum Beispiel darin, die Motorsimulation aus der DLL in die Testtreiber MC812Test und MC832Test zu verlegen. Damit könnte der XCTL-Quelltext entlastet und somit eine bessere Schichtenarchitektur in XCTL realisiert werden. Fortan könnte das normale XCTL-Treiberinterface auch für Simulationen genutzt werden,

[6] verwendet wird %SYSTEMROOT%, unter Windows 2000 üblicherweise “C:\WINNT“

ohne die Notwendigkeit, spezielle Interfaces für die Simulation, wie es momentan der Fall ist, implementieren zu müssen.

CVS Modul

Sämtliche treiberbezogene Dateien sind in dem CVS Modul XCTL_32_DRV zusammengefaßt. An dieser Stelle soll nun der Inhalt dieses Moduls dokumentiert werden.
Im dem CVS Modul befinden sich folgende Verzeichnisse:

Für die Quellen der echten Hardwaretreiber:

\DCBraunPSD
\DCGeneric
\DCRadicon
\MC812
\MC832

Für die Quellen der Testtreiber:

\DCBraunPSDTest
\DCGenericTest
\DCRadiconTest
\MC812Test
\MC832Test

Für die Quellen der Geräteklasseninstaller und Co-Installer (siehe Anhang B *Installationsumgebung*):

\DCx.Dll
\MC8x2.Dll

Für die Treiberdisketten mit Installationsscript und Treiberbinaries:

\DriverDisksDBG
\DriverDisksREL

Dabei ist die Dateistruktur der einzelnen Verzeichnisse ist wie folgt aufgebaut.

Treiberquellen	
Dateiname	**Beschreibung**
MC*.c	Treiberimplementierung Motor
MC*.h	Header Motor
DC*.c	Treiberimplementierung Detektor
DC*.h	Header Detektor
MC_IOCTL.h	IOCTL Definitionen und Parameterstrukturen Motor
DC_IOCTL.h	IOCTL Definitionen und Parameterstrukturen Detektor
MAKEFILE	vom DDK generierte Datei, darf nicht verändert werden
SOURCES	Treiberspezifische Scriptdatei für DDK Build Utility
driver.rc	Ressourcendatei, für alle Treiber bis auf #include identisch, dient im Wesentlichen der Erfüllung von Punkt 5. der Treiberspezifikation (siehe Treiberspezifikation)

Die Verzeichnisse der Geräteklasseninstaller und Co-Installer beinhalten je ein VC Projekt, mit dem eine DLL erstellt werden kann, die die für die Installation der Treiber notwendigen Installer beinhaltet.

Die eigentlichen Treiberdisketten befinden sich in den beiden \DriverDisk* Verzeichnissen. Dabei steht '*' für DBG (wie DeBuG) und REL (wie RELease). Beide unterscheiden sich in zwei Punkten voneinander:

1. die Treiber im Debugverzeichnis geben bei jeder ausgeführten Operation eine Zeichenkette an den Kerneldebugger ab
2. die Testtreiber sind nur im Debugverzeichnis vohanden und können nur über das dort befindliche Installationsscript installiert werden

Beide Verzeichnisse enthalten je ein Unterverzeichnis für die Detektortreiber (\DCx) und eines für die Motortreiber (\MC8x2). Diese Verzeichnisse werden als Treiberdisketten bezeichnet. Ihr gesamter Inhalt bildet ein Installationsmedium. Aus diesem Grunde darf diese Dateizusammenstellung für eine erfolgreiche Installation nicht verändert werden. Folgende Tabelle zeigt ein Installationsmedium (Treiberdiskette) im einzelnen:

Inhalt einer Treiberdiskette (Installationsmedium)	
Dateiname	**Beschreibung**
DC*.sys	Binärdateien der Windowstreiber für Detektoren
MC*.sys	Binärdateien der Windowstreiber für Motoren
DCx.inf	Installationsscript für Detectortreiber
MC8x2.inf	Installationsscript für Motortreiber
DCx.dll	Geräteklassen- und Co-Installer für Detektoren
MC8x2.dll	Geräteklassen- und Co-Installer für Motoren

Beschreibung Installationsumgebung

Die im Rahmen der 32-bit Portierung des XCTL-Systems entstandenen Treiber müssen, bevor sie vom XCTL-System genutzt werden können, erst einmal installiert werden. Das Hinzufügen von Gerätetreibern zu einem Windows-2000-System ist ein standardisierter Ablauf, unter der Voraussetzung, dass für einen Treiber mindestens ein Installationsscript existiert. Darüber hinaus kann auch ein Geräteklasseninstaller und / oder ein Co-Installer die Installation unterstützen.
Wenn ein Treiber vom System geladen werden soll, so muss dieser Treiber dem System bekannt sein. Um dies zu realisieren, existieren in der Systemregistrierung Schlüssel, die dafür bestimmt sind, derartige Informationen aufzunehemen. Da jedoch das direkte Manipulieren dieser Registryschlüssel sehr aufwendig und fehleranfällig wäre, hat Microsoft ein System entwickelt, welches es dem Programmierer erlaubt, mit Hilfe von Scriptdateien die Installation einfach und sicher zu steuern. Diese Scriptdateien mit der Dateierweiterung *.inf sind syntaktisch identisch mit den INI-Dateien. Semantisch sind sie jedoch wesentlich komplexer. Wesentlich zum Verständnis des Systems ist hierbei, dass innerhalb dieser *.inf Dateien Sektionen existieren, die sowohl das Kopieren der Dateien als auch das Initialisieren der entsprechenden Registryschlüssel steuern. Je installierbarem Treiber existiert eine eigene Sektion, so dass das Installationsverhalten für jeden Treiber individualisiert werden kann. Für Einzelheiten der Script Implementierung sei an dieser Stelle auf die entsprechenden Kommentare in den *.inf Dateien verwiesen.
Der eigentliche Installationsvorgang kann über zwei Wege aktiviert werden: zum einen über den Kontextdialog der INF-Datei im Windows Explorer: „Installieren“, zum anderen über den „ Hardware-Assistenten “ (siehe Anhang C – Werkzeuge) Die erste Möglichkeit wurde von uns explizit unterbunden bei der Entwicklung der Installationsscripte, da der Installationsvorgang unbedingt in Interaktion mit dem Benutzter geführt werden muss um benutzte Ressourcen von ihm zu erfragen, da das System aufgrund mangelnder PnP-Fähigkeiten der XCTL-Hardware nicht in der Lage ist, diese automatisch zu ermitteln.
An dieser Stelle soll nun kurz auf das Verwaltungskonzept für Geräte unter Windows eingegangen werden. Unter Windows 2000 erfolgt die Verwaltung der installierten Geräte (also Gerätetreiber) zentral über den Gerätemanager (siehe Anhang C – Werkzeuge). Änderungen an den Gerätekonfigurationen oder den Geräteressourcen sowie die Aktualisierung von Treibern können über dieses Programm gesteuert werden. Dabei werden installierte Geräte unter bestimmten

Gesichtspunkten zu Klassen zusammengefasst, um deren Verwaltung zu vereinfachen (z.B. durch die gemeinsame Nutzung bestimmter Bibliotheksressourcen). Eine solche Geräteklasse ist im Gerätemanager im Baumdiagramm als ein eigener Knoten mit eigenem Symbol gekennzeichnet. Bei der Entwicklung der Installation haben wir uns entschieden, die XCTL-Treiber nicht in eine allgemeine Geräteklasse einzufügen, sondern zwei neue Geräteklassen anzulegen: CDetectorControl und CMotorControl. Auch wenn sich diese beiden Geräteklassen stark ähneln, so ist die Aufteilung besonders mit Blick auf eine mögliche spätere Erweiterung des XCTL-Systems um weitere Hardware zu rechtfertigen. Um nun eine neue Geräteklasse dem System hinzuzufügen, ist es empfehlenwert laut Mircosoft [MSDN 03b], einen Geräteklasseninstaller einzusetzen. Dabei handelt es sich um eine Funktionsbibliothek in Form einer DLL, welche Routinen mit einem definierten Interface [MSDN 03b], [Oney 03] zu Verfügung stellt, die dazu dienen, Installation und Verwaltung einer Geräteklasse durch das Betriebssystem zu unterstützen. Diese Routinen werden ebenfalls via Installationsscript dem Betriebssystem bekannt gemacht, so dass der Entwickler die Möglichkeit hat, auf Installations- und Verwaltungsaufgaben für seine neue Geräteklasse Einfluss zu nehmen. Der Einfluss der Geräteklasseninstaller ist klassenglobal, d.h., alle zu einer Klasse gehörenden Treiber werden durch den Geräteklasseninstaller beeinflusst (zB. Icons, Propertypages). Im Falle der XCTL-Geräteklassen hat der Geräteklasseninstaller jedoch nur die eine Aufgabe zu erfüllen, nämlich ein spezielles Klassensymbol zur visuellen Identifikation der Geräteklasse hinzuzufügen. Dies ist zwar für das XCTL-System funktional völlig unerheblich, jedoch wird die Usability entscheidend verbessert, sowohl zur Entwicklung als auch im realen Einsatz der Treiber.
Im Gegensatz dazu hat der Co-Installer nur lokalen Einfuss auf genau ein Gerät einer bestimmten Klasse. Der von uns implementierte Co-Installer hat folgende Aufgaben (nur für Testtreiber, bei echten Hardwaretreibern hat der Co-Installer keine Funktion):

1. Während der Installation mit dem Hardwareassistenten muss auch bei den Testtreibern eine ID abgefragt werden. Da aber keine echte Hardware verwendet wird, kann der vom Installationsassistenten bereitgestellte Dialog zum Erfragen der Ressourcen nicht verwendet werden, weswegen der Co-Installer einen eigens dafür entworfenen Dialog anzuzeigen hat.

2. Nach erfolgreicher Installation hat der Co-Installer dafür zu sorgen, dass die zugewiesene ID vom Benutzer jederzeit geändert werden kann. Dafür muss eine eigene Propertypage erzeugt und dem Eigenschaftsdialog hinzugefügt werden. Die Gründe hierfür sind die gleichen wie unter 1.
3. Das Reinitialisieren der Testtreiber nach ID-Änderung ohne Systemneustart muss ebenfalls vom Co-Installer übernommen werden.

Im vorigen Abschnitt unter *CVS Modul* ist beschrieben, wo die Quellen und Projektdateien dieser DLLs zu finden sind. Implementierungsdetails sind den ausführlich kommentierten Quellen zu entnehmen.
Während der Installation muss, wie bereits erwähnt, die Controller ID erfragt werden. Dies geschieht bei den echten Hardwaretreibern über einen systemeigenen Standarddialog. Auch nach erfolgreicher Installation können die Ressourcen der Controller geändert werden. Dies ist über den Eigenschaftsdialog aus dem Kontextmenü des betroffenen Gerätes auf der Registerkarte „Ressourcen“ möglich. Auch die ID der Testtreiber kann auf diese Art geändert werden, jedoch mit dem Unterschied, dass kein Neustart nach der Änderung nötig ist und dass es sich bei diesen Eingabemasken nicht um systemeigene handelt, sondern um vom Co-Installer erstellte.
Des Weiteren ist anzumerken, dass bei echten Hardwaretreibern im Ressourcen-Dialog bereits auf eventuelle Ressourcen-Konflikte hingewiesen wird, die es selbstverständlich zu beachten gilt. Bei den Testtreibern jedoch ist der Benutzer dafür verantwortlich, keine IDs mehrfach zu vergeben, da sonst XCTL mit den Testtreibern nicht fehlerfrei kommunizieren kann.

Anhang B

Installationsanleitung für XCTL-Treiber für Entwickler

Im folgenden Abschnitt soll eine Schritt-für-Schritt-Anleitung zur Installation der XCTL- Treibersuite gegeben werden, unter den verschiedenen Möglichkeiten für Entwickler des XCTL-System. Im Anhang findet sich eine Anleitung für Anwender des XCTL-Systems, die ein komplettes System für den realen Einsatz selbst aufsetzen müssen.

1. Hardware-Assistent (über Systemsteuerung, System) starten

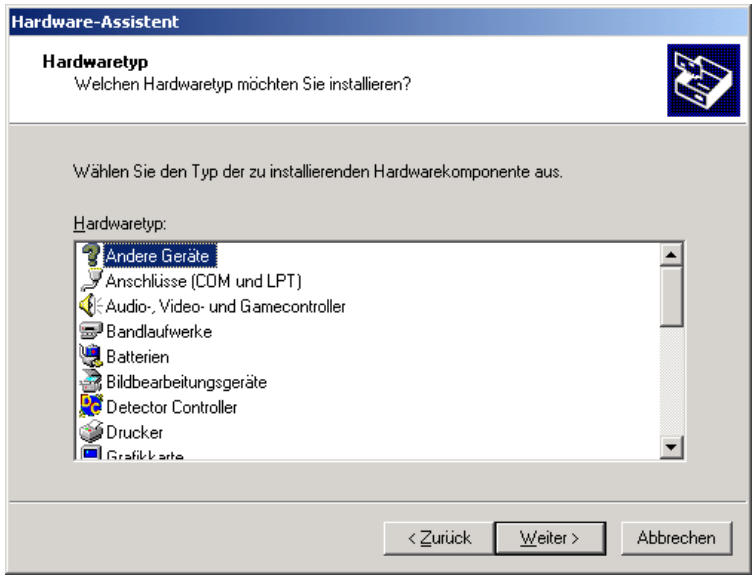
2. Dialog: Hardwareoption wählen
   -Option „Gerät hinzufügen“ auswählen
   -Button „Weiter“ anklicken

3. Warten, es werden keine neuen Geräte gefunden

4. Dialog: „Gerät wählen“
   -Option "Neues Gerät hinzufügen"
   -Button „Weiter“ anklicken

5. Dialog: „Suche nach neuen Hardwarekomponenten“
   -Option "Nein, die Hardwarekomponenten selbst in der Liste auswählen" anwählen
   -Button „Weiter“ anklicken

6. Dialog: „Hardwaretyp“
   -entweder
      Option "Andere Geräte" auswählen
   -oder
      wenn bereits schon einmal Geräte für XCTL installiert wurden (z.B. wenn eine weitere Treiberinstanz installiert werden soll bei zwei C-812 Motorkarten), kann die entsprechende Geräteklasse direkt angewählt werden, da sie dem System bereits bekannt ist, dabei entfällt Angabe der Treiberposition. (im Bild: Detector Controller)



7. Warten

8. Dialog: „Gerätetreiber auswählen“

-Button "Datenträger..." anklicken

-Button "Durchsuchen" anklicken und die Position des Treiberverzeichnisses mit Installationsscipt (*.inf) angeben

9. Dialog: „Gerätetreiber auswählen“

-Liste Modelle: in dieser Liste erscheinen nun, je nach dem, ob das Verzeichnis mit den Debugversionen (\DriverDisksDBG) oder das mit den Releaseversionen (\DriverDiskREL) ausgewählt wurde, die verfügbaren Treiber mit (in den Debugversionen) oder ohne (in den Releaseversionen) "Test" -Suffix. Davon abghängig, ob man einen echten Hardwaretreiber oder eine Testtreiber auswählt, ergeben sich zwei unterschiedliche Installationsszenarien:

a: echter Hardwaretreiber

b: Testtreiber

10a: Dialog: „Hardwareinstallation starten“

-der Hardwareassistent kann die benötigten Ressourcen nicht ermitteln, da es sich nicht um PnP-Hardware handelt. Darum erscheint eine entsprechende Meldung.

-Button „OK“ anklicken

Daraufhin öffnet sich ein Fenster, in welchem dann die Ressourcen für die entsprechende Controllerkarte ausgewählt werden müssen. Genau diese Ressourcen müssen dann auch auf der Controllerkarte mittels Dip-Switches oder Jumpern eingestellt werden. Wenn

Ressourcen ausgewählt werden, die bereits von anderer Hardware benutzt werden, wird vom Hardwarewizard darauf hingewiesen. In diesem Falle sind unbedingt konfliktfreie Ressourcen auszuwählen.

11a: Dialog: „Hardwareinstallation starten“

-es wird die Geräteklasse des aktuell zu installierenden Gerätes angezeigt

-Button „Weiter“

12a: Dialog: „Fertigstellen des Assistenten“

-die Installation kann nun beendet werden, wenn kein Fehler aufgetreten ist. Wenn die Ressourcen jetzt noch geändert werden sollen, kann dies durch einen Klick auf "Ressourcen..." noch getan werden, ansonsten

-Button "Fertigstellen“ klicken

13a: Hinweise:

-Die Installation ist nun beendet und der Rechner muss neu gestartet werden, um die Änderungen wirksam werden zu lassen. Wenn die Controllerkarten noch nicht eingebaut worden sind, so muss dies nun geschehen.

10b: Dialog: „Hardwareinstallation starten“

-Der Hardwarassistent erkennt, dass keine Ressourcen vom System zur Verfügung gestellt werden müssen, deshalb kann sofort mit "Weiter" fortgesetzt werden.

11b: Dialog: „Einstellen von Basisport/Basisadresse des Testtreibers“

-Dieser Dialog erscheint nur während der Installation eines Testtreibers. Er dient dazu die ID des Testtreibers einzustellen, damit XCTL mit dem Testtreiber kommunizieren kann. Der eingestellte Wert ist völlig beliebig, da keine echte Hardware dahinter steht. Der Dialog macht in der Combobox Vorgaben, welche übernommen werden können. Wichtig ist, keine ID mehrfach zu vergeben, da sonst XCTL nicht mit der richtigen Testtreiberinstanz kommunizieren wird. Im Falle der echten Hardwaretreiber muss man sich nicht selber darum kümmern, da daß System während der Installation (wie beschrieben unter 10a) auf Konflikte hinweisst.

-Button „Weiter“ anklicken

12b: Dialog: „Fertigstellen des Assistenten“

-wenn die Installation erfolgreich war, ist der Testtreiber sofort bereit, d.h., er kann sofort verwendet werden. Ein Button "Ressourcen..." existiert hier nicht, da keine echten Ressourcen vergeben wurden. Die ID kann im Gerätemanager jederzeit geändert werden, ohne dass der Rechner neu gestartet werden muss. Dazu verfährt man wie mit jedem anderen im Gerätemanager zu verwaltenden Gerät auch. (Eigenschaften aus Gerätekontextmenü, Ressourcen)

Bemerkung: Die XCTL-Treiber (mit Ausnahme der Testtreiber) führen bei jedem Systemstart einen Kommunikationstest mit der jeweiligen Hardware durch. Sollte ein solcher Test fehlschlagen, wird der zugehörige Treiber sofort wieder entladen, was daran zu erkennen ist, dass am entsprechenden Symbol im Gerätemanager ein Ausrufezeichen erscheint. In diesem Falle ist unter den angegebenen Ressourcen entweder keine Hardware installiert oder die Hardware ist defekt. In XCTL kommt es dann beim Start zu einer Fehlermeldung, dass der entsprechende Treiber nicht gefunden werden konnte.

Installationsanleitung für XCTL-Treiber für Anwender

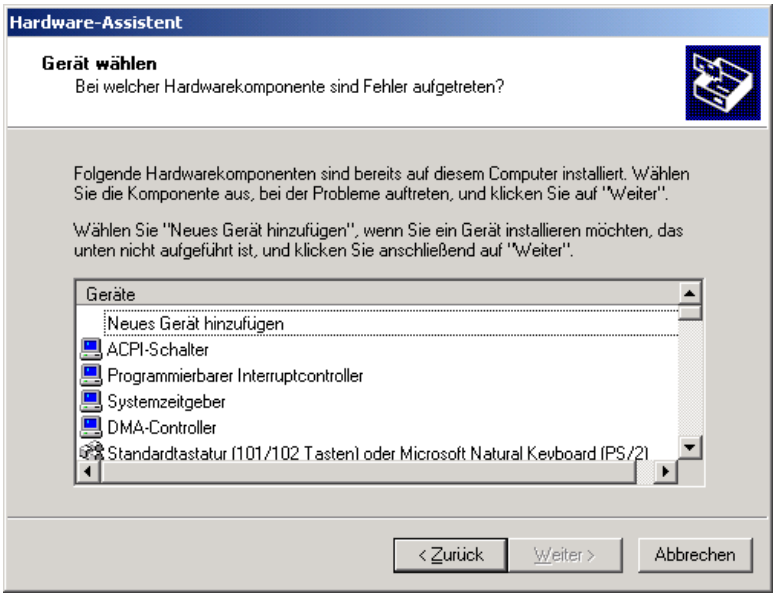
Im folgenden Abschnitt soll eine Schritt-für-Schritt Anleitung zur Installation der XCTL-Treibersuite gegeben werden. Diese Anleitung richtet sich an Anwender des XCTL-Systems, die ein komplettes System für den realen Einsatz aufsetzen müssen.

1. Hardware-Assistent (über Start, Einstellungen, Systemsteuerung, System) starten

2. Dialog: Hardwareoption wählen
   -Option „Gerät hinzufügen“ auswählen
   -Button „Weiter“ anklicken

3. Warten, es werden keine neuen Geräte gefunden

4. Dialog: „Gerät wählen“
   -Option "Neues Gerät hinzufügen"
   -Button „Weiter“ anklicken



5. Dialog: „Suche nach neuen Hardwarekomponenten“
   -Option "Nein, die Hardwarekomponenten selbst in der Liste auswählen" anwählen
   -Button „Weiter“ anklicken

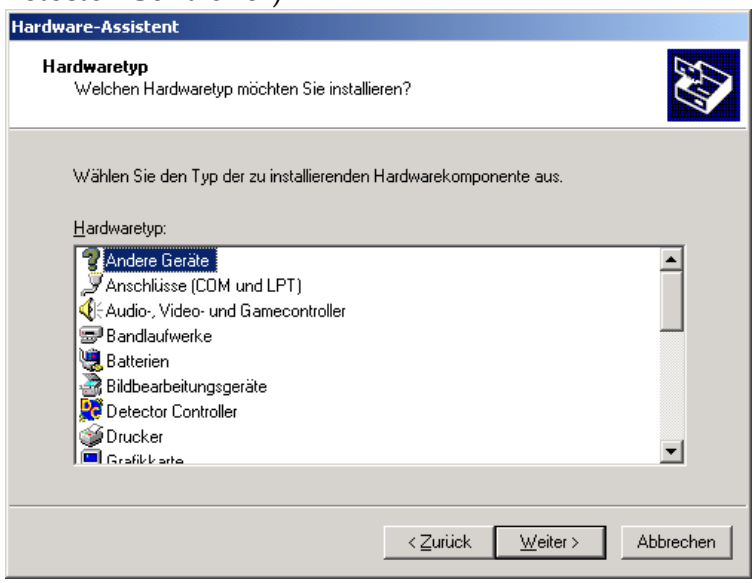
6. Dialog: „Hardwaretyp“

-entweder

Option "Andere Geräte" auswählen (wenn neue Treiber existieren, immer diesen Punkt wählen)

-oder

wenn bereits schon einmal Geräte für XCTL installiert wurden, kann die entsprechende Geräteklasse auch direkt angewählt werden (im Bild existiert bereits die Geräteklasse Detector Controller)



7. Warten

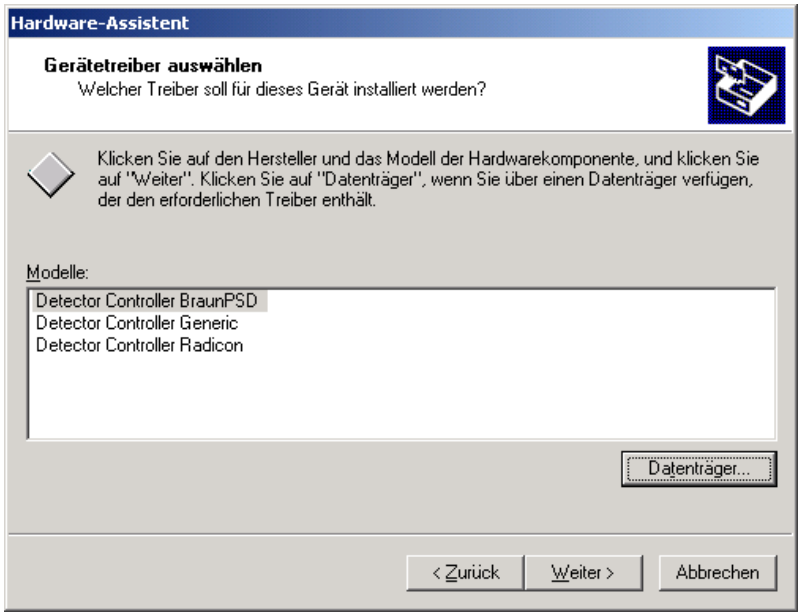
8. Dialog: „Gerätetreiber auswählen“

-Button "Datenträger..." anklicken

-Button "Durchsuchen" anklicken und die Position des Treiberverzeichnisses mit Installationsscipt (*.inf) angeben

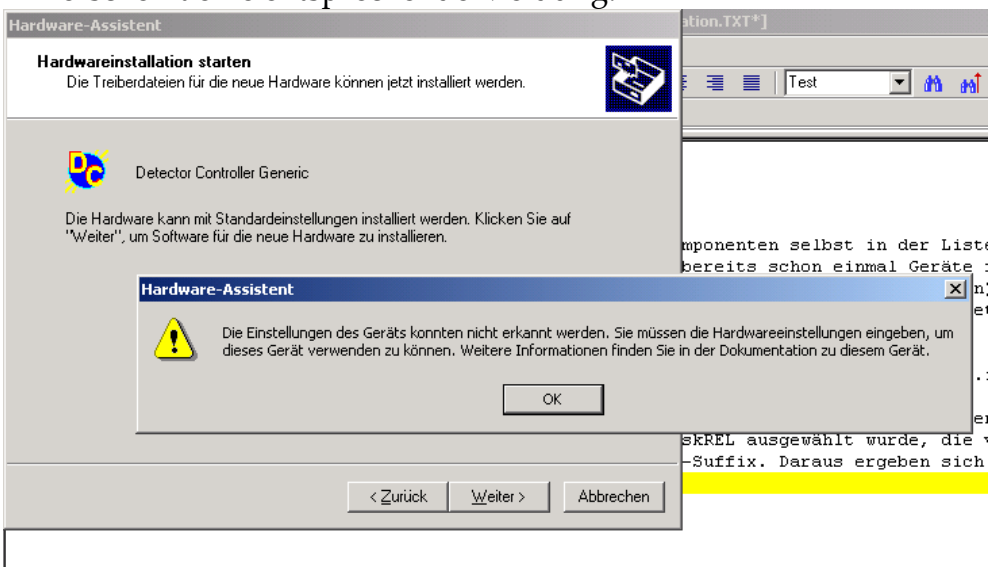
9. Dialog: „Gerätetreiber auswählen“

-Liste Modelle: in dieser Liste erscheinen nun die verfügbaren Treiber



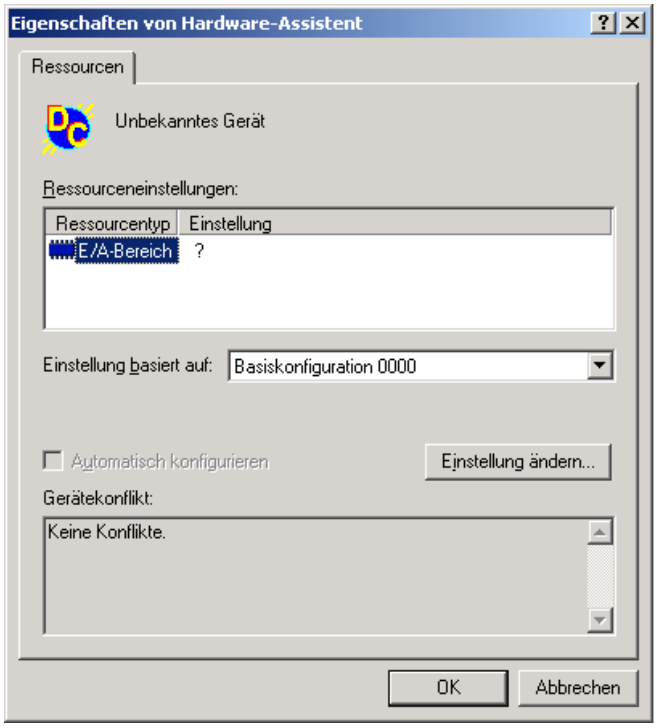
10: Dialog: „Hardwareinstallation starten“

-der Hardwareassistent kann die benötigten Ressourcen nicht ermitteln, da es sich nicht um PnP Hardware handelt. Darum erscheint eine entsprechende Meldung.



-Button „OK“ anklicken

Daraufhin öffnet sich ein Fenster, in welchem dann die Ressourcen für die entsprechende Controllerkarte ausgewählt werden müssen.



Genau diese Ressourcen müssen dann auch auf der Controllerkarte mittels Dip-Switches oder Jumpern eingestellt werden. Wenn Ressourcen ausgewählt werden, die bereits von anderer Hardware benutzt werden, wird vom Hardwarewizard darauf hingewiesen. In diesem Falle sind unbedingt konfliktfreie Ressourcen auszuwählen.

11: Dialog: „Hardwareinstallation starten“

-es wird die Geräteklasse des aktuell zu installierenden Gerätes angezeigt

-Button „Weiter“

12: Dialog: „Fertigstellen des Assistenten“

-die Installation kann nun beendet werden, wenn kein Fehler aufgetreten ist. Wenn die Ressourcen jetzt noch geändert werden sollen, kann dies durch einen Klick auf "Ressourcen..." noch getan werden, ansonsten

-Button "Fertigstellen“ klicken

13: Hinweise:

-Die Installation ist nun beendet und der Rechner muss neu gestartet werden, um die Änderungen wirksam werden zu lassen. Wenn die Controllerkarten noch nicht eingebaut worden sind, so muss dies nun geschehen.

Bemerkung: Die XCTL-Treiber führen bei jedem Systemstart einen Kommunikationstest mit der jeweiligen Hardware durch. Sollte ein solcher Test fehlschlagen, wird der zugehörige Treiber sofort wieder entladen, was daran zu erkennen ist, dass am entsprechenden Symbol im Gerätemanager ein Ausrufezeichen erscheint. In diesem Falle ist unter den angegebenen Ressourcen entweder keine Hardware installiert oder die Hardware ist defekt. In XCTL kommt es dann beim Start zu einer Fehlermeldung, dass der entsprechende Treiber nicht gefunden werden konnte.

Anhang C

Werkzeuge

Im gesamten Verlauf der Arbeit wurde von den Autoren eine Vielzahl von Hilfsprogrammen verwendet. Dieser Abschnitt stellt diese Werkzeuge dar. Dabei sind die Werkzeuge entsprechend ihrer Verwendung den einzelnen Phasen der Portierung zugeordnet (siehe Tabelle [C.1]). Dabei muss erwähnt werden, dass die Grenzen zwischen diesen Phasen in der Werkzeugbenutzung nicht immer eindeutig zu ziehen sind. Die Einordnung in die Portierungsphasen soll demzufolge als Hinweis auf die Fähigkeiten und Einsatzmöglichkeiten verstanden werden und nicht als enge unflexible Rahmenvorgabe für die Anwendung der Werkzeuge.

Verwendete Werkzeuge		
Name	**Kategorie**	**Einsatzzweck / Überwiegend verwendet bei:**
Doxygen	Quelltextdokumentations-werkzeug	Reverse Engineering, Analyse Hostsystem, Portierungsergebnis
W32Dasm	Disassembler	
BorlandC	Entwicklungsumgebung	
Porttool	statische Quelltextanalyse	
RC_Converter.exe	statische Konvertierung	Portierung
Visual Studio	Entwicklungsumgebung	Entwicklung
WinDbg	Debugging	
DebugView	Debugging	
DDK	Compiler, Funktionsbibliotheken	
Gerätemanager	Verwaltung	
Hardwareinstallations-assistent	Verwaltung	
ATOS	Testwerkzeug	Test
Verifier.exe	Testwerkzeug	

Tabelle [C.1] zeigt eine Übersicht der verwendeten Werkzeuge

Doxygen

Beschreibung:

Doxygen ist ein Werkzeug, mit dessen Hilfe aus einem Quelltext, der in einer bestimmten Programmiersprache vorliegt, ein Dokument angelegt werden kann, welches die Struktur dieses Quelltextes unter bestimmten, frei wählbaren Gesichtspunkten darstellt. Dabei können

zum Beispiel Querverweislisten über die Aufrufbeziehungen der Funktionen im Quelltext angelegt werden. Durch die strukturierte Darstellung in den Ausgabeformaten (u.a. html, rtf) kann der Quelltext leicht dokumentiert werden, da ein Quasi-Formular angelegt wird, welches vom Entwickler nur noch ausgefüllt zu werden braucht. Dabei erkennt Doxygen im Quelltext bereits vorhandene Kommentare automatisch und versucht diese, größtenteils sogar erfolgreich, entsprechend in die Dokumentation einzubauen.

Verwendung:

Doxygen wurde von den Autoren überwiegend für die Analyse des Ist-Zustandes von XCTL verwendet. Besonders die Crossreferencer-Funktionalität und die Fähigkeit die Klassenstruktur grafisch zu exportieren, waren dabei sehr hilfreich.

Weiterführende Informationen unter:

http://www.doxygen.org

W32Dasm

Beschreibung:

W32Dasm ist ein Dissassembler / Debugger. Das Programm läuft unter Win32 und ist in der Lage, sowohl Win16- als auch Win32-Code zu disassemblieren. Darüber hinaus ist es in der Lage, Import/Export-Beziehungen zu analysieren und Ressourcen zu extrahieren.

Verwendung:

W32Dasm wurde von den Autoren vorwiegend für die Analyse der ASA.DLL genutzt, um die mangelhafte Dokumentation der Programmierung des BraunPSD nachzuvollziehen. Des Weiteren diente W32Dasm zur Analyse der Architektur des 16-bit XCTL.

Weiterführende Informationen unter:

http://www.softnews.ro/public/cat/5/1/5-1-7.shtml

BorlandC

Beschreibung:

Hierbei handelt es sich um eine alte Entwicklungsumgebung für Windowsprogramme. Die 16-bit Version von XCTL wurde mit BorlandC erstellt.

Verwendung:

BorlandC Version 5.02 wurde für die Übersetzung der 16-bit XCTL-Variante verwendet, um mit Hilfe bestimmter Änderungen das

Verhalten dieser Version unter der alten Umgebung untersuchen zu können.

Weiterführende Informationen unter:

siehe Kapitel 3 *Analyse Entwicklungsumgebung*

http://info.borland.com/techpubs/borlandcpp/

Porttool

Beschreibung:

Im Zuge der Einführung von Win32 stellte Microsoft dieses Werkzeug zur Verfügung, um Entwicklern die Portierung ihres bereits existierenden, für Win16 ausgelegten Codes zu erleichtern. Es handelt sich im Wesentlichen nur um einen einfachen Texteditor mit einfachster Ersetzungsfunktiontionalität. Basierend auf einer Konfigurationsdatei, die Microsoft erstellt hat, gibt Porttool Hinweise, auf zu ersetzende Bibliotheksfunktionen.

Verwendung:

Der Einsatz dieses Werkzeuges ist aufgrund der einfachen Textersetzungsfunktionalität (keine regulären Suchbegriffe möglich) nicht in solch umfangreichen Projekten wie XCTL zu verwenden. Jedoch liefert die Konfigurationsdatei von Porttool konkrete Anhaltspunkte über Problempotential bei der Portierung in.

Weiterführende Informationen unter:

[Microsoft99b]

http://msdn.microsoft.com/

RC_Converter.exe

Beschreibung:

RC_Converter.exe dient der Konvertierung von Ressourcendateien mit Borland-spezifischen Erweiterungen zu rc-Dateien, welche von Visual Studio eingelesen werden können.

Verwendung:

Die Portierung der rc-Dateien wurde mit diesem Werkzeug unterstützt. Dadurch wurde der Aufwand des manuellen Portierens der rc-Dateien enorm reduziert, jedoch nicht völlig beseitigt.

Weiterführende Informationen unter:

RC_Converter.exe ist von Günther Reinecker.
reinecke@informatik.hu-berlin.de
[Projekt98]

Visual Studio 6

Beschreibung:

Hierbei handelt es sich u.a. um eine C++ Entwicklungsumgebung von Microsoft.

Verwendung:

Visual Studio 6 wurde im Rahmen der Portierung eingeführt und wird weiterhin die Grundlage der Weiterentwicklung von XCTL sein. Zu beachten ist, dass während der Portierung Visual Studio 6 mit Servicepack 5 verwendet wurde. Somit wird dringend dazu geraten, fortan mindestens dieses Servicepack zu verwenden.

Weiterführende Informationen unter:

[Microsoft 99b]
http://msdn.microsoft.com/

WinDbg

Beschreibung:

Bei WinDbg handelt es sich um einen extrem leistungsfähigen Debugger von Microsoft. Dieser steht zum freien Download bereit. Mit Hilfe dieses Debuggers ist es möglich, das gesamte System schon beim Hochfahren zu untersuchen, was besonders der Gerätetreiberentwicklung zugute kommt.

Verwendung:

WinDbg wurde von den Autoren dazu eingesetzt, die im Rahmen der Portierung erstellten Gerätetreiber zu debuggen. Auch die Installer-DLLs wurden mit WinDbg auf Fehler untersucht.

Weiterführende Informationen unter:

http://msdn.microsoft.com/

DebugView

Beschreibung:

Mit Hilfe von DebugView können Programmausgaben, welche an einen Kernelmodusdebugger gesendet wurden, angezeigt werden. Mit Hilfe spezieller Funktionen kann aus einem Programm eine solche Zeichenkette abgesetzt werden, um bestimmte Debuginformationen zur Laufzeit überwachen zu können. Im Normalfall wären dazu ein Kernelmodusdebugger wie WinDbg und ein zweiter PC nötig. Da aber in vielen Fällen die Debugfunktionalität nicht nötig ist, reicht zur Anzeige dieser Zeichenketten ein Programm ohne Debugfunktionalität völlig aus. Dieses kann dann auch auf dem zu untersuchenden System laufen, der zweite PC entfällt somit. Genau für diesen Fall ist DebugView bestens geeignet

Verwendung:

DebugView wurde von den Autoren ausschließlich für die Überwachung der im Rahmen der Portierung erstellten Gerätetreiber verwendet. Aufgrund der einfachen Handhabung dieses Werkzeuges ist es für die Weiterentwicklung von XCTL und den entsprechenden Treibern (Gerätetreiber und Testtreiber) sehr zu empfehlen.

Weiterführende Informationen unter:

http://www.sysinternals.com/ntw2k/freeware/debugview.shtml

DDK

Beschreibung:

DDK steht für **D**evice Driver **D**evelopment **K**it und wird von Microsoft im Rahmen des MSDN bereitgestellt, um Hardwareentwicklern eine Möglichkeit zu bieten, Gerätetreiber für ihre Hardware zu erstellen. Es handelt sich dabei um einen Satz Softwarewerkzeuge und einen speziellen Compiler, welche die Gerätetreibererstellung erst ermöglichen.

Verwendung:

Für die Erstellung der Gerätetreiber wurde das Windows-XP-DDK verwendet. Dieses DDK enthält auch ein aktuelles Windows-2000-DDK.

Weiterführende Informationen unter:

Das Windows-XP-DDK steht nicht mehr zum Download zur Verfügung, ist aber im MSDN Software Packet enthalten.
http://msdn.microsoft.com/

Gerätemanager

Beschreibung:

Der Gerätemanager ist ein Teil des Hardwareverwaltungsapparates von Windows 2000. Er ist in jeder Installation enthalten und dient der komfortablen Verwaltung der im System installierten Geräte und Gerätetreiber. Mit Hilfe des Gerätemanagers ist die Deinstallation, die Deaktivierung sowie die Veränderung der Ressourcen eines Treibers möglich.

Verwendung:

Der Gerätemanager wurde in Entwicklung und Test des portierten XCTL-Systems seiner Funktionen entsprechend verwendet.

Weiterführende Informationen unter:

Der Gerätemanager ist in der Hilfe von Windows 2000 dokumentiert.

Hardwareassistent

Beschreibung:

Der Hardwareassistent ist ein Teil des Hardwareverwaltungsapparates von Windows 2000 und in jeder Installation automatisch vorhanden. Mit seiner Hilfe können u.a. Gerätetreiber für solche Geräte installiert werden, die vom System nicht erkannt werden können, da sie nicht PnP-kompatibel sind.

Verwendung:

Der Hardwaresassistent wurde entsprechend seiner Funktionalität zum Installieren von Gerätetreibern während der gesamten Portierung verwendet. Auch zukünftig muss er verwendet werden, wenn ein XCTL-Gerätetreiber installiert werden soll, da die aktuelle XCTL-Hardware keine PnP-Fähigkeiten aufweist.

Weiterführende Informationen unter:

Der Hardwaresassistent ist in der Windows-2000-Hilfe dokumentiert.

Verifier.exe

Beschreibung:

Verifier.exe ist ein Testwerkzeug. Mittels Verifier.exe kann ein Geätetreiber auf Windows-2000-Konformität getestet werden. Verifier.exe ist Bestandteil jeder Windows-2000-Installation. Dieses Werkzeug ist von Microsoft entwickelt worden, um Gerätetreiber auf ihre Zuverlässigkeit im System zu untersuchen. Auch Microsoft verwendet dieses Werkzeug, wenn ein Hardwarehersteller einen seiner Gerätetreiber von Microsoft mit dem Windows-Logo-Test zertifizieren lassen will.

Verwendung:

Verifier.exe wurde in Zwischentests während der Entwicklungsphase der XCTL-Gerätetreiber und der eigentlichen Testphase der Portierung verwendet.

Weiterführende Informationen unter:

Der Hardwaresassistent ist unter Windows 2000 im System32-Verzeichnis der Windowsinstallation zu finden.

Siehe auch [Solomon 00]

ATOS

Beschreibung:

ATOS ist ein Testwerkzeug, mit Hilfe dessen oberflächenbasierte Testsequenzen erstellt und an einem zu testenden Programm ausgeführt werden können. Es wurde im Rahmen einer Diplomarbeit von Johann Letzel und Jens Hanisch entwickelt.

Verwendung:

ATOS wurde bereits auf die 16-bit Version von XCTL angesetzt. Auch die portierte XCTL-Version wurde teilweise mit ATOS getestet.

Weiterführende Informationen unter:

[ATOS 02]

[Projekt 98]

Anhang D

Tabellen

Ressourcenbedarf der Treiber		
XCTL-Treiber	**Ressourcentyp**	**Größe in Byte**
MC812	Memory Mapped I/O	801
MC832	Port I/O	2
DCGeneric	Port I/O	4
DCRadicon	Port I/O	2
DCBraunPSD	Port I/O	4

Tabelle [D.1]

Größe von Datentypen unter Win16 und Win32		
Datentyp	**Win16**	**Win32**
int	16 Bit	32 Bit
HANDLE	unsigned int, 16 Bit	unsigned int, 32 Bit
HWND, HPEN, HBRUSH	HANDLE	HANDLE
UINT	unsigned int, 16 Bit	unsigned int, 32 Bit
WORD	unsigned int, 16 Bit	unsigned int, 16 Bit
DWORD	unsigned int, 32 Bit	unsigned int, 32 Bit
LONG	signed int, 32 Bit	signed int, 32 Bit
NEAR Pointer	16 Bit	32 Bit
FAR Pointer	32 Bit	32 Bit

Tabelle [D.2.1], Quelle [Lauer 92]

Darstellungsformat der Windowsnachrichten				
	Win16		**Win32**	
Message	**wParam**	**lParam (lo,hi)**	**dwParam (lo,hi)**	**lParam**
WM_ACTIVE	state	minimized, hwnd	state, minimized	hwnd
WM_CHARTOITEM	char	pos, hwnd	char, pos	hwnd
WM_COMMAND	id	hwnd, cmd	id, cmd	hwnd
WM_CTLCOLOR*)	hdc	hwnd, type	hdc	hwnd
WM_MENUESELECT	cmd	flags, hmenu	cmd, flags	hmenu
WM_MENUECHAR	char	hmenue, fmenu	char, fmenu	hmenu
WM_PARENTNOTIFY	msg	id, hwndchild	msg, id	hwndchild
WM_HSCROLL	code	pos, hwnd	code, pos	hwnd
WM_VSCROLL	code	pos, hwnd	code, pos	hwnd

*) durch sieben neue Nachrichten ersetzt

Tabelle [D.2.2], Quelle [Lauer 92]

Bibliotheksfunktionen - Grafik	
Win16	**portable Version (Win16 und Win32)**
MoveTo	MoveToEx
OffsetViewportOrg	OffsetViewportOrgEx
OffsetWindowOrg	OffsetWindowOrgEx
ScaleViewportExt	ScaleViewportExtEx
ScaleWindowExt	ScaleWindowExtEx
SetBitmapDimension	SetBitmapDimensionEx
SetMetaFileBits	SetMetaFileBitsEx
SetVieportExt	SetVieportExtEx
SetWindowExt	SetWindowExtEx
SetWindowOrg	SetWindowOrgEx
GetAspectRatioFilter	GetAspectRatioFilterEx
GetBitmapDimension	GetBitmapDimensionEx
GetBrushOrg	GetBrushOrgEx
GetCurrentPosition	GetCurrentPositionEx
GetTextExtent	GetTextExtentPoint
GetTextExtentEx	GetTextExtentExPoint
GetViewportExt	GetViewportExtEx
GetViewportOrg	GetViewportOrgEx
GetWindowExt	GetWindowExtEx
GetWindowOrg	GetWindowOrgEx

Tabelle [D.3.1], Quelle [Microsoft 99b]

Bibliotheksfunktionen - Dialogfelder	
Win16	**portable Version (Win16 und Win32)**
DlgDirSelect	DlgDirSelectEx
DlgDirSelectComboBox	DlgDirSelectComboBoxEx

Tabelle [D.3.2], Quelle [Microsoft 99b]

Bibliotheksfunktionen – geänderte Funktionalität unter Win32
Funktion *)
GetActiveWindow
GetCapture
GetFocus
ReleaseCapture
SetActiveWindow
SetCapture
SetFocus

*) für weitergehende Informationen siehe [MSDN 03a]

Tabelle [D.3.3], Quelle [Microsoft 99b]

Portabilität von Bibliotheksfunktionen für Hardwarezugriff			
Funktion	**Win16**	**Win32 (Windows 9x/Me)**	**Win32 (Windows 2000/XP)**
inp	+	+	-
inpw	+	+	-
inpd	+	+	-
outp	+	+	-
outpw	+	+	-
outpd	+	+	-

Tabelle [D.4], Quelle [MSDN 03a]

existierende ATOS-Testsequenzen			
Testsequenz	**Anwendungsfall**	**Teilschritt**	**Hardware-nutzung**
Test_AE.1	Allgemeine Einstellungen	Ausführen des Dialogs „Allgemeine Einstellungen“	
Test_AE.2	Allgemeine Einstellungen	Anzeige des Dialogs „Über XCTL-Programm“	
Test_AJ.1	Automatische Justage	Konfiguration und Ausführen des Vorgangs „Automatische Justage“	ja
Test_ARS.1	Diffraktometrie / Reflektometrie ⇒ Area-Scan	Konfiguration des Vorgangs „Area-Scan“	
		Ausführen des Vorgangs „Area-Scan“ mit Messunterbrechung und -fortsetzung	ja
		Prüfen der Messwerte	
		Scankurven-Vergleich (zweite Messung)	
Test_ARS.2	Diffraktometrie / Reflektometrie ⇒ Area-Scan	Konfiguration des Vorgangs „Area-Scan“	
		Ausführen des Vorgangs „Area-Scan“	ja
		Datenbasis zerlegen	
		Datenbasis zusammensetzen	
Test_ARS.3	Diffraktometrie / Reflektometrie ⇒ Area-Scan	Impulsspektrum anzeigen	
		Detektor kalibrieren	ja
		Energiespektrum anzeigen	

Test_ARS.PD.1	Diffraktometrie / Reflektometrie ⇒ Area-Scan mit Protokollbuch	Konfiguration des Vorgangs „Area-Scan“	
		Protokollbuch ausfüllen	
		Ausführen des Vorgangs „Area-Scan“ mit Messunterbrechung und –fortsetzung	ja
		Prüfen der Messerwerte	
Test_AS.1	Ablaufsteuerung	Auswählen, Ausführen, Unterbrechen der Ausführung, Fortsetzen der Ausführung von Makros	
Test_D0.1	Detektornutzung (0-dimensionale)	Ausführen des Dialogs „Zähler-Konfiguration“ und Anzeige der Messwerte (digitale Anzeige)	ja
Test_D0.2	Detektornutzung (0-dimensionale)	Ausführen des Dialogs „Zähler-Konfiguration“ und Anzeige der Messwerte (Balken-Darstellung)	ja
Test_DM.1	Darstellung von Messwerten	Laden und Darstellung gespeicherter Messwerte als Kurve (Area-Scan)	
		Darstellung als Raw-Matrix	
		Modifikation der Darstellungsoptionen	
		Darstellung als RL-Bitmap	
Test_DM.2	Darstellung von Messwerten	Laden und Darstellung gespeicherter Messwerte als Raw-Matrix (Area-Scan)	
		Bearbeitung der Messdaten	

		Laden und Darstellen der bearbeiteten Messdaten als Kurve (Line-Scan)	
Test_HWB.1	Halbwertsbreite messen	Ausführen des Vorgangs „Halbwertsbreite messen“	ja
Test_LS.1	Diffraktometrie / Reflektometrie ⇒ Line-Scan	Konfiguration des Vorgangs „Step-Scan“	
		Ausführen des Vorgangs „Line-Scan“ mit Messunterbrechung und –fortsetzung	ja
		Scankurven-Vergleich (zweite Messung)	
Test_LS.2	Diffraktometrie / Reflektometrie ⇒ Line-Scan	Konfiguration des Vorgangs „Continuous-Scan“	
		Ausführen des Vorgangs „Line-Scan“ mit Messunterbrechung und –fortsetzung	ja
Test_LS.3	Diffraktometrie / Reflektometrie ⇒ Line-Scan	Konfiguration des Vorgangs „Step-Scan“	
		Ausführen des Vorgangs „Line-Scan“ mit Messunterbrechung und –fortsetzung (dynamische Schrittweite)	ja
Test_LS.PD.1	Diffraktometrie / Reflektometrie ⇒ Line-Scan mit Protokollbuch	Konfiguration des Vorgangs „Step-Scan“	
		Konfiguration des Vorgangs „Continuous-Scan“	
		Protokollbuch ausfüllen	
		Ausführen des Vorgangs „Line-Scan“ mit Messunterbrechung und –fortsetzung	ja
Test_MJ.1	Manuelle Justage	Positionieren des Motors DF (neuer Winkel, relative Null	ja

		setzen, relative Null aufheben)	
Test_MJ.2	Manuelle Justage	Positionieren des Motors Tilt mittels Cursor-Tasten und Scrollbar (Schritt-Betrieb)	ja
Test_MJ.3	Manuelle Justage	Positionieren des Motors Absorber mittels Cursor-Tasten und Scrollbar (Fahr-Betrieb)	ja
Test_MJ.4	Manuelle Justage	gleichzeitiges Positionieren der Motoren Absorber und Kollimator (neuer Winkel)	ja
Test_MJN.1	Neue Manuelle Justage		ja
Test_MJN.2	Neue Manuelle Justage		ja
Test_MJN.3	Neue Manuelle Justage		ja
Test_MJN.4	Neue Manuelle Justage		ja
Test_MJN.5	Neue Manuelle Justage		ja
Test_MJN.6	Neue Manuelle Justage	Ausführen des Vorgangs „Halbwertsbreite messen“	ja
Test_MS.1	Motorsteuerung	Ausführen des Vorgangs „Referenzpunktlauf“ für Motor DF	ja
		Ausführen des Vorgangs „Direkte Steuerung...“ für Motor DF	ja
		Ausführen des Vorgangs „Referenzpunktlauf“ für Motor Tilt	ja
		Ausführen des Vorgangs „Direkte Steuerung...“ für Motor Tilt	ja
Test_MS.2	Motorsteuerung	Optimieren des Motors Azimutal Rot	ja
		Optimieren des Motors DF	ja

Test_MS.3	Motorsteuerung	Ausführen des Vorgangs „Referenzpunktlauf“ für Motor Omega	ja
		Konfiguration des Motors Theta (absoluter Nullpunkt, End-Schalter)	ja
		Positionieren des Motors Theta (neuer Winkel)	ja
Test_PD.1	Protokollbuch Diffraktometrie / Reflektometrie (deutsch)	Protokollbuch anlegen	
		Protokollbuch ausfüllen und bearbeiten	
Test_PD.2	Protokollbuch Diffraktometrie / Reflektometrie (deutsch)	Protokollbuch anlegen	
		Protokollbuch ausfüllen und bearbeiten	
Test_PT.1	Protokollbuch Topographie (deutsch)	Protokollbuch anlegen	
		Protokollbuch ausfüllen und bearbeiten	
Test_PT.2	Protokollbuch Topographie (deutsch)	Protokollbuch anlegen	
		Protokollbuch ausfüllen und bearbeiten	
Test_TP.1	Topographie	Konfiguration des Vorgangs „Topographie“	
		Ausführen des Vorgangs „Startposition einstellen“	ja
		Ausführen des Vorgangs „Regelung starten“	ja
Test_TP.2	Topographie	Konfiguration des Vorgangs „Topographie“	
		Ausführen des Vorgangs „Startposition einstellen“	ja
		Ausführen des Vorgangs „Regelung starten“	ja

Test_TP.HWB .PT.1	Topographie mit Halbwertsbreite messen und Protokollbuch	Konfiguration des Vorgangs „Topographie“	
		Ausführen des Vorgangs „Halbwertsbreite messen“ (Manuelle Justage (Alt))	ja
		Ausführen des Vorgangs „Halbwertsbreite messen“ (Manuelle Justage NEU)	ja
		Ausführen des Vorgangs „Startposition einstellen“	ja
		Ausführen des Vorgangs „Regelung starten“	ja
		Protokollbuch ausfüllen	
Test_TPG.1	Getrennte Topographie	Konfiguration des Vorgangs „Topographie“	
		Ausführen des Vorgangs „Startposition einstellen“	ja
		Ausführen des Vorgangs „Regelung starten“	ja
Test_TPG.2	Getrennte Topographie	Konfiguration des Vorgangs „Topographie“	
		Ausführen des Vorgangs „Startposition einstellen“	ja
		Ausführen des Vorgangs „Regelung starten“	ja

Tabelle [D.5]

Anhang E

Klassendiagramme der im Rahmen der Portierung geänderten Klassen

(1) Klasse TC_812ISA

TC_812ISA
- DWORD **dwBaseAddr** - TC_812Controller * **Hardware**
+ **TC_812ISA** (void) # int **ExecuteCmd** (char *) # BOOL **CheckBoardOk** (void) # BOOL **SetHome** (void) # void **StartCheckScan** (void) # BOOL **IsMoveFinish** (void) # BOOL **_GetPosition** (long *) # BOOL **_GetFailure** (long *) - int **PutChar** (const char) - char **GetChar** (void)

(2) Klasse TC_832

TC_832
+ void(* **lpfnLimitWatch**)(UINT, UINT, DWORD, DWORD, DWORD) # WORD **wKI** # WORD **wKL** # WORD **wKP** # WORD **wKD** - WORD **wBaseAddr** - int **nOnBoardId** - BYTE **cConfig** - WORD **wSample** - TC_832Controller * **Hardware** - DWORD **ControllerIndex** - UINT **nEvent** - const WORD **CmdRegister** - const WORD **DataRegister** - WORD **baddr** - WORD **raddr**
+ **TC_832** (void) + BOOL **StopLimitWatch** (void) + BOOL **StartLimitWatch** (void) + void **OptimizingDlg** (void)

int **SaveSettings** (BOOL) # BOOL **ActivateFilterParameters** (void) # BOOL **CheckBoardOk** (void) # BOOL **ActivateDrive** (void) # void **StartCheckScan** (void) # int **Reset** (void) - long **Drive628** (BYTE, WORD, long) - int **ExecuteCmd** (LPSTR) - BOOL **IsMoveFinish** (void) - BOOL **IsLimitHit** (void) - BOOL **IsIndexArrived** (void) - BOOL **StartToIndex** (long &) - BOOL **StopDrive** (BOOL) - BOOL **SetLimit** (DWORD) - BOOL **SetVelocity** (DWORD) - DWORD **GetVelocity** (void) - BOOL **SetHome** (void) - BOOL **SetAcceleration** (DWORD) - DWORD **GetAcceleration** (void) - WORD **GetStatus** (void) - BOOL **_GetPosition** (long *) - BOOL **_GetFailure** (long *) - BOOL **MoveRelative** (long) - BOOL **MoveAbsolute** (long)

(4) Klasse TGenericDetector

TGenericDetector
- TGenericController * **Hardware** - int **nBaseAddr** - float **fTimeCorrection**
+ **TGenericDetector** (int id) + **~TGenericDetector** () + BOOL **Initialize** (void) + TDetectorType **GetDetectorType** () const + int **MeasureStart** (void) + int **MeasureStop** (void) + int **PollDetector** (void) + void **LookUp** (LPCSTR Section) # BOOL **_SetParameters** (void)

(3) Klasse TRadicon

TRadicon
- double **dRealTime** - int **Rdd** - int **Rcc** - TRange< WORD > **DacThresh** - WORD **wHighVoltage** - TRadiconController * **Hardware** - static const TRange< WORD > **DacThreshLimits** - static const TRange< WORD > **VoltageLimits** - static TRadiconController * **TheEventHardware**
+ **TRadicon** (int id) + **~TRadicon** () + BOOL **Initialize** (void) + BOOL **InitializeEvent** (HWND, int) + TDetectorType **GetDetectorType** () const + WORD **GetHighVoltage** () const + void **SetHighVoltage** (WORD) + void **SetDacThresh** (WORD, WORD) + const TRange< WORD > & **GetDacThresh** () const + int **MeasureStart** (void) + int **MeasureStop** (void) + int **PollDetector** (void) + BOOL **RunSpecificParametersDlg** (void) + BOOL **HasSpecificParametersDlg** (void) const + void CALLBACK **EventHandler** (UINT, UINT, DWORD, DWORD, DWORD) - BOOL **_SetParameters** (void) - BOOL **SetSound** (void) - void **LoadDetectorSettings** () - void **SaveDetectorSettings** () const

(5) Klasse TBraunPsd

TBraunPsd
+ int **nErrorCode** + BOOL **bSetError** + int **nHVRegelung_OK** - int **nBaseAddr** - UINT **uMuxTimeDet1** - LONG **PositionsDatenHeader** [16] - UINT **uEnergyScale** - UINT **uPositionScale** - UINT **uEnergyHigh** - UINT **uEnergyLow** - LONG **lPositionStop** - BOOL **bAbbruchmitShutter** - int **nDeathTime** - BOOL **bRatemeter** - BOOL **bRealLifeTime** - LONG **lMeasTime** - HGLOBAL **hReadBuf** - int **nDelayFast** - int **nDelaySlow** - UINT **uCBufferLength** - int **nReadBufItems** - int **nHVControl_OK** - TBraunPsdController * **Hardware** - static BYTE **echo** [30][16] - static BYTE **befehl** [30][16]
+ **TBraunPsd** (int id) + **~TBraunPsd** (void) + BOOL **Initialize** (void) + int **PsdInit** (void) + int **PsdStart** (void) + int **PsdReadOut** (THowReadOutPsd) + int **GetBufferSize** (void) + BOOL **SetEnergyRange** (UINT, UINT) + void **GetEnergyRange** (UINT &ler, UINT &her) const + int **PsdStop** (void) + int **GetChannelNumber** () const + TDetectorType **GetDetectorType** () const # int **BuildOperation** (BYTE *, BYTE *, int) # int **LoadHexFile** (void) # void **ResetDelayTime** (void) # BYTE **konvert** (char) # int **SynchronHexFile** (BYTE &, BYTE &) # int **Look_till_BaseAddr1** (int) # void **LoadDetectorSettings** () # void **SaveDetectorSettings** () const

Anhang F

Auswahl PC-Hardware

Aus den unter Kapitel 3 *Analyse Hardwareumgebung* zusammengetragenen Informationen ergibt sich, dass bei der Erneuerung der Computerhardware im Zuge der XCTL -Portierung ein Kompromiss zu finden ist zwischen den Anforderungen, die Windows 2000 als Zielbetriebssystem stellt und denen der anwendungsspezifischen Hardware der Meßplätze.
Eine wichtige Voraussetzung für den Betrieb der anwendungsspezifischen Hardware der Messplätze stellt die ISA-Schnittstelle der Erweiterungskarten dar. Es müssen bei einer minimalen Konfiguration für einen Messplatz mindestens ein Detektor und vier Motoren ansteuerbar sein. Dies bedeutet im Falle der Verwendung der Motorsteuerkarte C-812, dass mindestens zwei ISA-Steckplätze vorhanden sein müssen. Wenn Motorsteuerkarten vom Typ C-832 zum Einsatz kommen sollen, so müssen mindestens drei ISA-Steckplätze frei sein. Um also minimale Flexibilität für den Einsatz der anwendungsspezifischen Hardware der Messplätze zu gewährleisten, werden als Mindestanzahl drei ISA-Steckplätze festgelegt. Jedoch ist dieser veraltete PC-Erweiterungsbus mittlerweile fast vollständig vom Consumer-Markt verschwunden. In seltenen Fällen werden noch Mainboards mit einem einzelnen ISA-Steckplatz angeboten, das Gros der aktuell verfügbaren Mainboards besitzt keinen ISA-Bus mehr. Damit scheidet die Beschaffung aktueller Mainboards der Consumer-Klasse aus.

Für die Lösung dieses Problems bieten sich zwei Möglichkeiten an:

(1) Verwendung ausrangierter PCs der HU-Berlin
(2) Beschaffung spezieller Industriemainboards
Recherchen der Autoren haben ergeben, dass der ISA-Erweiterungsbus in Industrieanwendungen noch weit verbreitet ist. Einige Hersteller von Mainboards haben sich darauf spezialisiert diesen Markt zu bedienen. Als Hinweis für weitere Recherchen sei hier beispielsweise die Firma ITOX genannt (www.itox.com). Sie liefert Mainboards für Intel Pentium III Prozessoren mit bis zu fünf ISA-Steckplätzen im ATX-Formfaktor.

Zum aktuellen Zeitpunkt hat man sich im Institut für Physik für die erste Variante entschieden, da die Mindestanforderungen von Windows 2000 (siehe Tabelle [3.4]) damit problemlos erfüllt werden können, was zugleich kostenschonend und umweltfreundlich ist.

Anhang G

Testsequenzen der dedizierten Hardwaretests

Test_AJ.H.1	
1	#"Skriptdatei für den Testfall AJ.H.1"
2	#"Anwendungsfall Automatische Justage (Radicon SCSCS)"
3	#"24.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK",FORCE
12	COPY,ENV,"TOPO_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	COPY,REF,"JUSTAGE.LOG.REF",TGT,"JUSTAGE.LOG.REF"
14	DELETE,TGT,"Justage.log",IFEXISTS
15	#"########## Testsequenz"
16	START,""
17	WAIT,7000
18	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
18	WINDOWEXISTS,"Zähler-Konfiguration",YES
20	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Radicon","Detektorauswahl"
21	ACTION,"Zähler-Konfiguration",CHECKBOX,UNCHECK,"Sound"
22	ACTION,"Zähler-Konfiguration",CHECKBOX,CHECK,"Anzeigen"
23	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
24	WINDOWEXISTS,"Zähler-Konfiguration",NO
25	ACTION,MAIN,MENU,CLICK,"Ausführen","Automatische Justage..."
26	WINDOWEXISTS,"Automatische Justage",YES
27	WINDOWEXISTS,"Zähler:",YES
28	ACTION,"Automatische Justage",CHECKBOX,CHECK,"Logdatei"
29	ACTION,"Automatische Justage",EDITBOX,EDIT,"0.1","Toleranz"
30	ACTION,"Automatische Justage",EDITBOX,EDIT,"3","Durchläufe"
31	ACTION,"Automatische Justage",CHECKBOX,UNCHECK,"maximale Intensitätsdiff."
32	ACTION,"Automatische Justage",EDITBOX,EDIT,"3","Anz.Intensitätsmessung"
33	ACTION,"Automatische Justage",EDITBOX,EDIT,"-39.58","Beugung Fein"
34	ACTION,"Automatische Justage",EDITBOX,EDIT,"20","Anz.Intensitätsmessung"
35	ACTION,"Automatische Justage",EDITBOX,EDIT,"0.00","Kollimator"
36	ACTION,"Automatische Justage",BUTTON,CLICK,"Start der automatischen Justage"
37	TEST,"Automatische Justage",BUTTON,DISABLED,"Start der automatischen Justage"
38	#WAIT,210000
39	MESSAGE,"OK, wenn bereit!"
40	TEST,"Automatische Justage",BUTTON,ENABLED,"Start der automatischen Justage"
41	ACTION,"Zähler:",WINDOW,CLOSE
42	WINDOWEXISTS,"Zähler:",NO
43	ACTION,"Automatische Justage",BUTTON,CLICK,"Beenden"
44	WINDOWEXISTS,"Automatische Justage",NO
45	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
46	WAIT,5000
47	#LAUNCH,BIN,"DATADIFF.EXE","JUSTAGE.LOG JUSTAGE.LOG.REF",FOREVER,0,"DataDiff.log"
48	#"########## Nachbereitung"
49	CLEANUP
50	DELETE,TGT,"JUSTAGE.LOG.REF"
51	COPY,TGT,"JUSTAGE.LOG",TGT,"AJH1.LOG"
52	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"

53	DELETE,TGT,"XCONTROL.BAK"
54	DELETE,TGT,"Justage.log"

Test_ARS.H.1	
1	#"Skriptdatei für den Testfall ARS.H.2"
2	#"Anwendungsfall Diffraktometrie/Reflektometrie - AreaScan (Omega2Theta)"
3	#"24.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"DEVELOP.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"AREA_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Öffnen","AreaScan-Fenster"
17	WINDOWEXISTS,"Areascan",YES
18	ACTION,"Areascan",MENU,CLICK,"Setup zum AreaScan..."
19	WINDOWEXISTS,"Einstellungen AreaScan",YES
20	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"36.700","Omega Minimum"
21	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"36.800","Omega Maximum"
22	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"0.005","Omega Schritt"
23	ACTION,"Einstellungen AreaScan",COMBOBOX,SELECT,"Braun","Detektorauswahl"
24	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"1","Meßzeit"
25	ACTION,"Einstellungen AreaScan",COMBOBOX,SELECT,"Kein Monitor","Monitor-Detektor"
26	ACTION,"Einstellungen AreaScan",CHECKBOX,UNCHECK,"Absorber benutzen"
27	ACTION,"Einstellungen AreaScan",CHECKBOX,UNCHECK,"Theta-Achse fixieren"
28	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"67.900","Theta Minimum"
29	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"2","Theta Relation"
30	ACTION,"Einstellungen AreaScan",CHECKBOX,UNCHECK,"Akkumuliert"
31	ACTION,"Einstellungen AreaScan",CHECKBOX,CHECK,"Nur am Ende"
32	ACTION,"Einstellungen AreaScan",CHECKBOX,CHECK,"Kontinuierlich sichern"
33	ACTION,"Einstellungen AreaScan",CHECKBOX,CHECK,"Bei Beenden speichern"
34	ACTION,"Einstellungen AreaScan",EDITBOX,EDIT,"C:\\","Sicherungs-Verzeichnis"
35	ACTION,"Einstellungen AreaScan",BUTTON,CLICK,"Ok"
36	WINDOWEXISTS,"Einstellungen AreaScan",NO
37	ACTION,"Areascan",MENU,CLICK,"AreaScan starten"
38	HANDLEMESSAGEBOX,"Start AreaScan",YES
39	WINDOWEXISTS,"Sichern unter ...",YES
40	MESSAGE,"Dateiname AREA1.PSD eingeben/auswählen und SPEICHERN drücken."
41	WINDOWEXISTS,"Sichern unter ...",NO
42	HANDLEMESSAGEBOX,"Meldung",NO
43	WAIT,45000
44	HANDLEMESSAGEBOX,"Meldung",OK
45	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
46	WINDOWEXISTS,"Manuelle Justage",YES
47	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Omega","Aktueller Antrieb"
48	ACTION,"Manuelle Justage",EDITBOX,SET,"36.7544","Neuer Winkel"
49	WAIT,3000
50	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Theta","Aktueller Antrieb"
51	ACTION,"Manuelle Justage",EDITBOX,SET,"68.000","Neuer Winkel"
52	WAIT,10000
53	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"

54	WINDOWEXISTS,"Manuelle Justage",NO
55	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
56	WAIT,5000
57	#"########## Nachbereitung"
58	CLEANUP
59	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
60	DELETE,TGT,"XCONTROL.BAK"
61	COPY,TGT,"STANDARD.BAK",TGT,"STANDARD.MAK"
62	DELETE,TGT,"STANDARD.BAK"

Test_D0.H.1	
1	#"Skriptdatei für den Testfall D0.H.1"
2	#"Anwendungsfall Detektornutzung - 0-dimensionale (Radicon SCSCS)"
3	#"24.07.2003"
5	#"########## Vorbereitung"
6	EXISTS,TGT,"XCONTROL.INI"
7	EXISTS,TGT,"HARDWARE.INI"
8	EXISTS,TGT,"STANDARD.MAK"
9	EXISTS,TGT,"TESTDEV.DAT"
10	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
11	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
12	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
13	COPY,ENV,"TOPO_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
14	COPY,REF,"D0H1.LOG.REF",TGT,"DEVICE.LOG.REF"
15	DELETE,TGT,"device0.log",IFEXISTS
16	#"########## Testsequenz"
17	START,""
18	WAIT,7000
19	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
20	WINDOWEXISTS,"Zähler-Konfiguration",YES
21	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Radicon","Detektorauswahl"
22	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"0.50","Zeit-Begrenzung"
23	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"20000","Impuls-Begrenzung"
24	ACTION,"Zähler-Konfiguration",CHECKBOX,UNCHECK,"Sound"
25	ACTION,"Zähler-Konfiguration",CHECKBOX,CHECK,"Anzeigen"
26	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
27	WINDOWEXISTS,"Zähler-Konfiguration",NO
28	ACTION,"Zähler:",MENU,CLICK,"Digitale Anzeige"
29	ACTION,"Zähler:",MENU,CLICK,"Log-File"
30	WAIT,12000
31	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
32	WINDOWEXISTS,"Zähler-Konfiguration",YES
33	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Radicon","Detektorauswahl"
34	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"1.0","Zeit-Begrenzung"
35	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
36	WINDOWEXISTS,"Zähler-Konfiguration",NO
37	WAIT,12000
38	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
39	WINDOWEXISTS,"Zähler-Konfiguration",YES
49	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Radicon","Detektorauswahl"
41	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"40000","Impuls-Begrenzung"
42	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
43	WINDOWEXISTS,"Zähler-Konfiguration",NO
44	WAIT,12000
45	ACTION,"Zähler:",MENU,CLICK,"Log-File"
46	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
47	WAIT,5000

48	#"########## Nachbereitung"
49	CLEANUP
50	COPY,TGT,"DEVICE0.LOG",TGT,"D0H1.LOG"
51	DELETE,TGT,"DEVICE.LOG.REF"
52	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
53	DELETE,TGT,"XCONTROL.BAK"

Test_D0.H.2	
1	#"Skriptdatei für den Testfall D0.H.2"
2	#"Anwendungsfall Detektornutzung - 0-dimensionale (Generic SCSCS)"
3	#"24.07.2003"
5	#"########## Vorbereitung"
6	EXISTS,TGT,"XCONTROL.INI"
7	EXISTS,TGT,"HARDWARE.INI"
8	EXISTS,TGT,"STANDARD.MAK"
9	EXISTS,TGT,"TESTDEV.DAT"
10	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
11	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
12	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
13	COPY,ENV,"TOPO_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
14	COPY,REF,"D0H1.LOG.REF",TGT,"DEVICE.LOG.REF"
15	DELETE,TGT,"device1.log",IFEXISTS
16	#"########## Testsequenz"
17	START,""
18	WAIT,7000
19	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
20	WINDOWEXISTS,"Zähler-Konfiguration",YES
21	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Generic","Detektorauswahl"
22	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"0.50","Zeit-Begrenzung"
23	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"20000","Impuls-Begrenzung"
24	ACTION,"Zähler-Konfiguration",CHECKBOX,UNCHECK,"Sound"
25	ACTION,"Zähler-Konfiguration",CHECKBOX,CHECK,"Anzeigen"
26	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
27	WINDOWEXISTS,"Zähler-Konfiguration",NO
28	ACTION,"Zähler:",MENU,CLICK,"Digitale Anzeige"
29	ACTION,"Zähler:",MENU,CLICK,"Log-File"
30	WAIT,12000
31	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
32	WINDOWEXISTS,"Zähler-Konfiguration",YES
33	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Generic","Detektorauswahl"
34	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"1.0","Zeit-Begrenzung"
35	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
36	WINDOWEXISTS,"Zähler-Konfiguration",NO
37	WAIT,12000
38	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
39	WINDOWEXISTS,"Zähler-Konfiguration",YES
49	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Generic","Detektorauswahl"
41	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"40000","Impuls-Begrenzung"
42	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
43	WINDOWEXISTS,"Zähler-Konfiguration",NO
44	WAIT,12000
45	ACTION,"Zähler:",MENU,CLICK,"Log-File"
46	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
47	WAIT,5000
48	#"########## Nachbereitung"
49	CLEANUP
50	COPY,TGT,"DEVICE1.LOG",TGT,"D0H2.LOG"

51	DELETE,TGT,"DEVICE.LOG.REF"
52	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
53	DELETE,TGT,"XCONTROL.BAK"

Test_D1.H.1	
1	#"Skriptdatei für den Testfall D1.H.1"
2	#"Anwendungsfall Detektornutzung - 1-dimensionale (Braun-PSD)"
3	#"24.07.2003"
5	#"########## Vorbereitung"
6	EXISTS,TGT,"XCONTROL.INI"
7	EXISTS,TGT,"HARDWARE.INI"
8	EXISTS,TGT,"STANDARD.MAK"
9	EXISTS,TGT,"TESTDEV.DAT"
10	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
11	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
12	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
13	COPY,ENV,"AREA_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
14	COPY,REF,"D1H1.LOG.REF",TGT,"DEVICE.LOG.REF"
15	DELETE,TGT,"device0.log",IFEXISTS
16	#"########## Testsequenz"
17	START,""
18	WAIT,7000
19	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
20	WINDOWEXISTS,"Zähler-Konfiguration",YES
21	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Braun","Detektorauswahl"
22	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"0.50","Zeit-Begrenzung"
23	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"2000","Impuls-Begrenzung"
24	ACTION,"Zähler-Konfiguration",CHECKBOX,UNCHECK,"Sound"
25	ACTION,"Zähler-Konfiguration",CHECKBOX,CHECK,"Anzeigen"
26	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
27	WINDOWEXISTS,"Zähler-Konfiguration",NO
28	ACTION,"Zähler:",MENU,CLICK,"Digitale Anzeige"
29	ACTION,"Zähler:",MENU,CLICK,"Log-File"
30	WAIT,12000
31	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
32	WINDOWEXISTS,"Zähler-Konfiguration",YES
33	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Braun","Detektorauswahl"
34	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"1.0","Zeit-Begrenzung"
35	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
36	WINDOWEXISTS,"Zähler-Konfiguration",NO
37	WAIT,12000
38	ACTION,MAIN,MENU,CLICK,"Einstellungen","Detektoren","Detektoren..."
39	WINDOWEXISTS,"Zähler-Konfiguration",YES
49	ACTION,"Zähler-Konfiguration",COMBOBOX,SELECT,"Braun","Detektorauswahl"
41	ACTION,"Zähler-Konfiguration",EDITBOX,EDIT,"4000","Impuls-Begrenzung"
42	ACTION,"Zähler-Konfiguration",BUTTON,CLICK,"Ok"
43	WINDOWEXISTS,"Zähler-Konfiguration",NO
44	WAIT,12000
45	ACTION,"Zähler:",MENU,CLICK,"Log-File"
46	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
47	WAIT,5000
48	#"########## Nachbereitung"
49	CLEANUP
50	COPY,TGT,"DEVICE0.LOG",TGT,"D1H1.LOG"
51	DELETE,TGT,"DEVICE.LOG.REF"
52	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
53	DELETE,TGT,"XCONTROL.BAK"

Test_HWB.H.1	
1	#"Skriptdatei für den Testfall HWB.H.1"
2	#"Anwendungsfall Halbwertsbreite messen - Radicon SCSCS"
3	#"24.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"TOPO_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Beugung Fein","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",BUTTON,CLICK,"Halbwertsbreite messen"
20	TEST,"Manuelle Justage",BUTTON,TEXT,"&Messung abbrechen","Halbwertsbreite messen"
21	#WAIT,210000
22	MESSAGE,"OK, wenn bereit!"
23	HANDLEMESSAGEBOX,"Information",OK
24	TEST,"Manuelle Justage",BUTTON,TEXT,"&Halbwertsbreite messen","Halbwertsbreite messen"
25	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
26	WINDOWEXISTS,"Manuelle Justage",NO
27	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
28	WAIT,5000
29	#"########## Nachbereitung"
30	CLEANUP
31	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
32	DELETE,TGT,"XCONTROL.BAK"

Test_LS.H.1	
1	#"Skriptdatei für den Testfall LS.H.1"
2	#"Anwendungsfall Diffraktometrie/Reflektometrie - LineScan (Radicon SCSCS)"
3	#"24.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"TOPO_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Öffnen","LineScan-Fenster"
17	WINDOWEXISTS,"LineScan",YES
18	ACTION,"LineScan",MENU,CLICK,"Setup ContinuousScan..."
19	WINDOWEXISTS,"Einstellungen ContinuousScan",YES

20	ACTION,"Einstellungen ContinuousScan",COMBOBOX,SELECT,"Beugung Fein","Scan-Achse"
21	ACTION,"Einstellungen ContinuousScan",EDITBOX,EDIT,"-21.370","Startpunkt"
22	ACTION,"Einstellungen ContinuousScan",EDITBOX,EDIT,"21.370","Endpunkt"
23	ACTION,"Einstellungen ContinuousScan",EDITBOX,EDIT,"0.2","Bereichsgröße"
24	ACTION,"Einstellungen ContinuousScan",COMBOBOX,SELECT,"Radicon","Sensor"
25	ACTION,"Einstellungen ContinuousScan",EDITBOX,EDIT,"1.0","Meßzeit"
26	ACTION,"Einstellungen ContinuousScan",CHECKBOX,CHECK,"Bei Beenden speichern"
27	ACTION,"Einstellungen ContinuousScan",EDITBOX,EDIT,"C:\\","Sicherungs-Verzeichnis"
28	ACTION,"Einstellungen ContinuousScan",BUTTON,CLICK,"Parameter aktualisieren"
29	TEST,"Einstellungen ContinuousScan",STATIC,NUM,"0.5","Geschwindigkeit"
30	TEST,"Einstellungen ContinuousScan",STATIC,NUM,"213",TOL,1,"Meßpunkte"
31	ACTION,"Einstellungen ContinuousScan",BUTTON,CLICK,"Ok"
32	WINDOWEXISTS,"Einstellungen ContinuousScan",NO
33	ACTION,"LineScan",MENU,CLICK,"Scan starten"
34	HANDLEMESSAGEBOX,"Start Scan",YES
35	WINDOWEXISTS,"Sichern unter ...",YES
36	MESSAGE,"Dateiname LINESCAN1.CRV eingeben/auswählen und SPEICHERN drücken."
37	WINDOWEXISTS,"Sichern unter ...",NO
38	HANDLEMESSAGEBOX,"Meldung",NO
39	WAIT,50000
40	#MESSAGE,"OK, wenn bereit!"
41	HANDLEMESSAGEBOX,"Meldung",OK
42	ACTION,"LineScan",WINDOW,CLOSE
43	WINDOWEXISTS,"LineScan",NO
44	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
45	WAIT,5000
46	#"########## Nachbereitung"
47	CLEANUP
48	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
49	DELETE,TGT,"XCONTROL.BAK"

Test_MJ.H.1.1	
1	#"Skriptdatei für den Testfall MJ.H.1.1"
2	#"Anwendungsfall Manuelle Justage - Direktbetrieb (C-812 BoardID 4)"
3	#"18.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Y","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"0.350","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"0","Neuer Winkel"
21	MESSAGE,"Position 0.000 anfahren - OK, wenn bereit!"
22	TEST,"Manuelle Justage",STATIC,NUM,"0",TOL,0.001,"Winkel"
23	ACTION,"Manuelle Justage",EDITBOX,SET,"-6.990","Neuer Winkel"
24	WAIT,9500
25	TEST,"Manuelle Justage",STATIC,NUM,"-6.990",TOL,0.001,"Winkel"

26	ACTION,"Manuelle Justage",EDITBOX,SET,"6.990","Neuer Winkel"
27	WAIT,18000
28	TEST,"Manuelle Justage",STATIC,NUM,"6.990",TOL,0.001,"Winkel"
29	ACTION,"Manuelle Justage",BUTTON,CLICK,"Relative Null setzen"
30	TEST,"Manuelle Justage",BUTTON,ENABLED,"Relative Null aufheben"
31	TEST,"Manuelle Justage",STATIC,NUM,"0.0","Winkel"
32	TEST,"Manuelle Justage",EDITBOX,NUM,"0.0","Neuer Winkel"
33	ACTION,"Manuelle Justage",EDITBOX,SET,"0.160","mit V="
34	ACTION,"Manuelle Justage",EDITBOX,SET,"-9.52","Neuer Winkel"
35	WAIT,25500
36	TEST,"Manuelle Justage",STATIC,NUM,"-9.52",TOL,0.001,"Winkel"
37	ACTION,"Manuelle Justage",BUTTON,CLICK,"Relative Null aufheben"
38	TEST,"Manuelle Justage",BUTTON,DISABLED,"Relative Null aufheben"
39	TEST,"Manuelle Justage",STATIC,NUM,"-2.53",TOL,0.001,"Winkel"
40	ACTION,"Manuelle Justage",EDITBOX,SET,"0.0","Neuer Winkel"
41	WAIT,7500
42	TEST,"Manuelle Justage",STATIC,NUM,"0.000",TOL,0.001,"Winkel"
43	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
44	WINDOWEXISTS,"Manuelle Justage",NO
45	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
46	WAIT,5000
47	#"########## Nachbereitung"
48	CLEANUP
49	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
50	DELETE,TGT,"XCONTROL.BAK"
51	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
52	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MJ.H.1.2	
1	#"Skriptdatei für den Testfall MJ.H.1.2"
2	#"Anwendungsfall Manuelle Justage - Direktbetrieb (C-832 BoardID 0)"
3	#"18.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Theta","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"7.529","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"220.000","Neuer Winkel"
21	MESSAGE,"Position 220.000 anfahren - OK, wenn bereit!"
22	TEST,"Manuelle Justage",STATIC,NUM,"220.000",TOL,0.001,"Winkel"
23	ACTION,"Manuelle Justage",EDITBOX,SET,"210.001","Neuer Winkel"
24	WAIT,21500
25	TEST,"Manuelle Justage",STATIC,NUM,"210.001",TOL,0.001,"Winkel"
26	ACTION,"Manuelle Justage",EDITBOX,SET,"339.999","Neuer Winkel"
27	WAIT,248000
28	TEST,"Manuelle Justage",STATIC,NUM,"339.999",TOL,0.001,"Winkel"

29	ACTION,"Manuelle Justage",BUTTON,CLICK,"Relative Null setzen"
30	TEST,"Manuelle Justage",BUTTON,ENABLED,"Relative Null aufheben"
31	TEST,"Manuelle Justage",STATIC,NUM,"0.0","Winkel"
32	TEST,"Manuelle Justage",EDITBOX,NUM,"0.0","Neuer Winkel"
33	ACTION,"Manuelle Justage",EDITBOX,SET,"3.160","mit V="
34	ACTION,"Manuelle Justage",EDITBOX,SET,"-9.52","Neuer Winkel"
35	WAIT,44500
36	TEST,"Manuelle Justage",STATIC,NUM,"-9.52",TOL,0.001,"Winkel"
37	ACTION,"Manuelle Justage",BUTTON,CLICK,"Relative Null aufheben"
38	TEST,"Manuelle Justage",BUTTON,DISABLED,"Relative Null aufheben"
39	TEST,"Manuelle Justage",STATIC,NUM,"330.479",TOL,0.001,"Winkel"
40	ACTION,"Manuelle Justage",EDITBOX,SET,"325.0","Neuer Winkel"
41	WAIT,26000
42	TEST,"Manuelle Justage",STATIC,NUM,"325.000",TOL,0.001,"Winkel"
43	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
44	WINDOWEXISTS,"Manuelle Justage",NO
45	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
46	WAIT,5000
47	#"########## Nachbereitung"
48	CLEANUP
49	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
50	DELETE,TGT,"XCONTROL.BAK"
51	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
52	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MJ.H.2.1	
1	#"Skriptdatei für den Testfall MJ.H.2.1"
2	#"Anwendungsfall Manuelle Justage - Schrittbetrieb (C-812 BoardID 4)"
3	#"19.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Y","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"0.350","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"0","Neuer Winkel"
21	MESSAGE,"Position 0.000 anfahren - OK, wenn bereit!"
22	TEST,"Manuelle Justage",STATIC,NUM,"0",TOL,0.001,"Winkel"
23	ACTION,"Manuelle Justage",RADIOBUTTON,CHECK,"Schritt-Betrieb"
24	ACTI2ON,"Manuelle Justage",EDITBOX,SET,"1.00","mit D="
25	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
26	LOOP,5
27	KEYBOARD,RIGHT,0
28	WAIT,2500
29	ENDLOOP
30	TEST,"Manuelle Justage",STATIC,NUM,"5.00",TOL,0.001,"Winkel"
31	TEST,"Manuelle Justage",EDITBOX,NUM,"5.00",TOL,0.001,"Neuer Winkel"

32	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
33	COMPARE,startpos,NUM,VAR,endpos,LSS
34	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
35	ACTION,"Manuelle Justage",EDITBOX,SET,"0.15","mit D="
36	LOOP,10
37	KEYBOARD,LEFT,0
38	WAIT,1500
39	ENDLOOP
40	TEST,"Manuelle Justage",STATIC,NUM,"3.5",TOL,0.001,"Winkel"
41	TEST,"Manuelle Justage",EDITBOX,NUM,"3.5",TOL,0.001,"Neuer Winkel"
42	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
43	COMPARE,startpos,NUM,VAR,endpos,GRT
44	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
45	LOOP,10
46	ACTION,"Manuelle Justage",HSCROLLBAR,RIGHT,0,"Scrollbar"
47	WAIT,1500
48	ENDLOOP
49	TEST,"Manuelle Justage",STATIC,NUM,"5.0",TOL,0.002,"Winkel"
50	TEST,"Manuelle Justage",EDITBOX,NUM,"5.0",TOL,0.002,"Neuer Winkel"
51	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
52	COMPARE,startpos,NUM,VAR,endpos,LSS
53	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
54	ACTION,"Manuelle Justage",EDITBOX,SET,"1.00","mit D="
55	LOOP,5
56	ACTION,"Manuelle Justage",HSCROLLBAR,LEFT,0,"Scrollbar"
57	WAIT,2500
58	ENDLOOP
59	TEST,"Manuelle Justage",STATIC,NUM,"0.00",TOL,0.002,"Winkel"
60	TEST,"Manuelle Justage",EDITBOX,NUM,"0.00",TOL,0.002,"Neuer Winkel"
61	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
62	COMPARE,startpos,NUM,VAR,endpos,GRT
63	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
64	WINDOWEXISTS,"Manuelle Justage",NO
65	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
66	WAIT,5000
67	#"########## Nachbereitung"
68	CLEANUP
69	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
70	DELETE,TGT,"XCONTROL.BAK"
71	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
72	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MJ.H.2.2	
1	#"Skriptdatei für den Testfall MJ.H.2.2"
2	#"Anwendungsfall Manuelle Justage - Schrittbetrieb (C-832 BoardID 0)"
3	#"19.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""

15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Theta","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"7.529","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"220.000","Neuer Winkel"
21	MESSAGE,"Position 220.000 anfahren - OK, wenn bereit!"
22	TEST,"Manuelle Justage",STATIC,NUM,"220.000",TOL,0.001,"Winkel"
23	ACTION,"Manuelle Justage",RADIOBUTTON,CHECK,"Schritt-Betrieb"
24	ACTION,"Manuelle Justage",EDITBOX,SET,"1.00","mit D="
25	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
26	LOOP,5
27	KEYBOARD,RIGHT,0
28	WAIT,4500
28	ENDLOOP
29	TEST,"Manuelle Justage",STATIC,NUM,"225.00",TOL,0.001,"Winkel"
30	TEST,"Manuelle Justage",EDITBOX,NUM,"225.00",TOL,0.001,"Neuer Winkel"
31	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
32	COMPARE,startpos,NUM,VAR,endpos,LSS
33	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
34	ACTION,"Manuelle Justage",EDITBOX,SET,"0.15","mit D="
35	LOOP,10
36	KEYBOARD,LEFT,0
37	WAIT,2000
38	ENDLOOP
39	TEST,"Manuelle Justage",STATIC,NUM,"223.5",TOL,0.001,"Winkel"
40	TEST,"Manuelle Justage",EDITBOX,NUM,"223.5",TOL,0.001,"Neuer Winkel"
41	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
42	COMPARE,startpos,NUM,VAR,endpos,GRT
43	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
44	LOOP,10
45	ACTION,"Manuelle Justage",HSCROLLBAR,RIGHT,0,"Scrollbar"
46	WAIT,2000
47	ENDLOOP
48	TEST,"Manuelle Justage",STATIC,NUM,"225.0",TOL,0.001,"Winkel"
49	TEST,"Manuelle Justage",EDITBOX,NUM,"225.0",TOL,0.001,"Neuer Winkel"
50	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
51	COMPARE,startpos,NUM,VAR,endpos,LSS
52	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
53	ACTION,"Manuelle Justage",EDITBOX,SET,"1.00","mit D="
54	LOOP,5
55	ACTION,"Manuelle Justage",HSCROLLBAR,LEFT,0,"Scrollbar"
56	WAIT,4500
57	ENDLOOP
58	TEST,"Manuelle Justage",STATIC,NUM,"220.00",TOL,0.001,"Winkel"
59	TEST,"Manuelle Justage",EDITBOX,NUM,"220.00",TOL,0.001,"Neuer Winkel"
60	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
61	COMPARE,startpos,NUM,VAR,endpos,GRT
62	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
63	WINDOWEXISTS,"Manuelle Justage",NO
64	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
65	WAIT,5000
66	#"########## Nachbereitung"
67	CLEANUP
68	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
69	DELETE,TGT,"XCONTROL.BAK"
70	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
71	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MJ.H.3.1	
1	#"Skriptdatei für den Testfall MJ.H.3.1"
2	#"Anwendungsfall Manuelle Justage - Fahrbetrieb (C-812 BoardID 4)"
3	#"20.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Y","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"0.350","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"0","Neuer Winkel"
21	MESSAGE,"Position 0.000 anfahren - OK, wenn bereit!"
22	ACTION,"Manuelle Justage",RADIOBUTTON,CHECK,"Fahren"
23	ACTION,"Manuelle Justage",EDITBOX,SET,"0.350","mit V="
24	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
25	KEYBOARD,RIGHT,5000
26	WAIT,1000
27	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
28	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
29	COMPARE,winkel,NUM,VAR,neuerwinkel
30	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
31	COMPARE,startpos,NUM,VAR,endpos,LSS
32	ACTION,"Manuelle Justage",EDITBOX,SET,"0.100","mit V="
33	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
34	ACTION,"Manuelle Justage",HSCROLLBAR,LEFT,5000,"Scrollbar"
35	WAIT,1000
36	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
37	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
38	COMPARE,winkel,NUM,VAR,neuerwinkel
39	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
40	COMPARE,startpos,NUM,VAR,endpos,GRT
41	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
42	ACTION,"Manuelle Justage",HSCROLLBAR,RIGHT,5000,"Scrollbar"
43	WAIT,1000
44	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
45	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
46	COMPARE,winkel,NUM,VAR,neuerwinkel
47	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
48	COMPARE,startpos,NUM,VAR,endpos,LSS
49	ACTION,"Manuelle Justage",EDITBOX,SET,"0.350","mit V="
50	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
51	KEYBOARD,LEFT,5000
52	WAIT,1000
53	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
54	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
55	COMPARE,winkel,NUM,VAR,neuerwinkel

56	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
57	COMPARE,startpos,NUM,VAR,endpos,GRT
58	TEST,"Manuelle Justage",STATIC,NUM,"0.000",TOL,0.01,"Winkel"
59	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
60	WINDOWEXISTS,"Manuelle Justage",NO
61	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
62	WAIT,5000
63	#"########## Nachbereitung"
64	CLEANUP
65	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
66	DELETE,TGT,"XCONTROL.BAK"
67	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
68	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MJ.H.3.2	
1	#"Skriptdatei für den Testfall MJ.H.3.2"
2	#"Anwendungsfall Manuelle Justage - Fahrbetrieb (C-832 BoardID 0)"
3	#"20.07.2003"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Theta","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"7.529","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"220.000","Neuer Winkel"
21	MESSAGE,"Position 220.000 anfahren - OK, wenn bereit!"
22	ACTION,"Manuelle Justage",RADIOBUTTON,CHECK,"Fahren"
23	ACTION,"Manuelle Justage",EDITBOX,SET,"7.0588","mit V="
24	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
25	KEYBOARD,RIGHT,5000
26	WAIT,2000
27	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
28	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
29	COMPARE,winkel,NUM,VAR,neuerwinkel
30	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
31	COMPARE,startpos,NUM,VAR,endpos,LSS
32	ACTION,"Manuelle Justage",EDITBOX,SET,"1.00","mit V="
33	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
34	ACTION,"Manuelle Justage",HSCROLLBAR,LEFT,5000,"Scrollbar"
35	WAIT,1000
36	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
37	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
38	COMPARE,winkel,NUM,VAR,neuerwinkel
39	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
40	COMPARE,startpos,NUM,VAR,endpos,GRT
41	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
42	ACTION,"Manuelle Justage",HSCROLLBAR,RIGHT,5000,"Scrollbar"

43	WAIT,1000
44	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
45	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
46	COMPARE,winkel,NUM,VAR,neuerwinkel
47	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
48	COMPARE,startpos,NUM,VAR,endpos,LSS
49	ACTION,"Manuelle Justage",EDITBOX,SET,"7.0588","mit V="
50	READ,"Manuelle Justage",HSCROLLBAR,POSITION,startpos,"Scrollbar"
51	KEYBOARD,LEFT,5000
52	WAIT,2000
53	READ,"Manuelle Justage",STATIC,NUM,winkel,"Winkel"
54	READ,"Manuelle Justage",EDITBOX,NUM,neuerwinkel,"Neuer Winkel"
55	COMPARE,winkel,NUM,VAR,neuerwinkel
56	READ,"Manuelle Justage",HSCROLLBAR,POSITION,endpos,"Scrollbar"
57	COMPARE,startpos,NUM,VAR,endpos,GRT
58	TEST,"Manuelle Justage",STATIC,NUM,"220.000",TOL,0.01,"Winkel"
59	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
60	WINDOWEXISTS,"Manuelle Justage",NO
61	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
62	WAIT,5000
63	#"########## Nachbereitung"
64	CLEANUP
65	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
66	DELETE,TGT,"XCONTROL.BAK"
67	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
68	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MJ.H.1.1	
1	#"Skriptdatei für den Testfall MS.H.1.1"
2	#"Anwendungsfall Motorsteuerung - Direkte Steuerung (C-812 BoardID 4)"
3	#"20.12.2002"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,5000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Y","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"0.350","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"0","Neuer Winkel"
21	MESSAGE,"Position 0.000 anfahren - OK, wenn bereit!"
22	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
23	WINDOWEXISTS,"Manuelle Justage",NO
24	ACTION,MAIN,MENU,CLICK,"Einstellungen","Antriebe","Direkte Steuerung..."
25	WINDOWEXISTS,"Verfahren nach Encoder-Position",YES
26	ACTION,"Verfahren nach Encoder-Position",COMBOBOX,SELECT,"Y","Motor"
27	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"0",TOL,0,"Neue Position"
28	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"0",TOL,0,"Position"
29	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,startpos,"Scrollbar"

30	ACTION,"Verfahren nach Encoder-Position",EDITBOX,SET,"10000","Neue Position"
31	WAIT,4500
32	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"10000","Neue Position"
33	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"10000","Position"
34	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,endpos,"Scrollbar"
35	COMPARE,startpos,NUM,VAR,endpos,LSS
36	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,startpos,"Scrollbar"
37	ACTION,"Verfahren nach Encoder-Position",EDITBOX,SET,"-2500","Neue Position"
38	WAIT,5500
39	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"-2500","Neue Position"
40	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"-2500","Position"
41	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,endpos,"Scrollbar"
42	COMPARE,startpos,NUM,VAR,endpos,GRT
43	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,startpos,"Scrollbar"
44	ACTION,"Verfahren nach Encoder-Position",EDITBOX,SET,"0","Neue Position"
45	WAIT,2500
46	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"0","Neue Position"
47	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"0","Position"
48	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,endpos,"Scrollbar"
49	COMPARE,startpos,NUM,VAR,endpos,LSS
50	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
51	WAIT,5000
52	#"########## Nachbereitung"
563	CLEANUP
54	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
55	DELETE,TGT,"XCONTROL.BAK"
56	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
57	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MS.H.1.2	
1	#"Skriptdatei für den Testfall MS.H.1.2"
2	#"Anwendungsfall Motorsteuerung - Direkte Steuerung (C-832 BoardID 0)"
3	#"20.12.2002"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,7000
16	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
17	WINDOWEXISTS,"Manuelle Justage",YES
18	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Theta","Aktueller Antrieb"
19	ACTION,"Manuelle Justage",EDITBOX,SET,"7.529","mit V="
20	ACTION,"Manuelle Justage",EDITBOX,SET,"220.000","Neuer Winkel"
21	MESSAGE,"Position 220.000 anfahren - OK, wenn bereit!"
22	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
23	WINDOWEXISTS,"Manuelle Justage",NO
24	ACTION,MAIN,MENU,CLICK,"Einstellungen","Antriebe","Direkte Steuerung..."
25	WINDOWEXISTS,"Verfahren nach Encoder-Position",YES
26	ACTION,"Verfahren nach Encoder-Position",COMBOBOX,SELECT,"Theta","Motor"
27	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"3168000",TOL,0,"Neue Position"

28	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"3168000",TOL,0,"Position"
29	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,startpos,"Scrollbar"
30	ACTION,"Verfahren nach Encoder-Position",EDITBOX,SET,"3468000","Neue Position"
31	WAIT,42500
32	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"3468000",TOL,2,"Neue Position"
33	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"3468000",TOL,2,"Position"
34	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,endpos,"Scrollbar"
35	#COMPARE,startpos,NUM,VAR,endpos,LSS
36	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,startpos,"Scrollbar"
37	ACTION,"Verfahren nach Encoder-Position",EDITBOX,SET,"3300000","Neue Position"
38	WAIT,24500
39	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"3300000",TOL,2,"Neue Position"
40	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"3300000",TOL,2,"Position"
41	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,endpos,"Scrollbar"
42	#COMPARE,startpos,NUM,VAR,endpos,GRT
43	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,startpos,"Scrollbar"
44	ACTION,"Verfahren nach Encoder-Position",EDITBOX,SET,"3390000","Neue Position"
45	WAIT,14500
46	TEST,"Verfahren nach Encoder-Position",EDITBOX,NUM,"3390000",TOL,2,"Neue Position"
47	TEST,"Verfahren nach Encoder-Position",STATIC,NUM,"3390000",TOL,2,"Position"
48	READ,"Verfahren nach Encoder-Position",HSCROLLBAR,POSITION,endpos,"Scrollbar"
49	#COMPARE,startpos,NUM,VAR,endpos,LSS
50	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
51	WAIT,5000
52	#"########## Nachbereitung"
53	CLEANUP
54	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
55	DELETE,TGT,"XCONTROL.BAK"
56	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
57	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MS.H.2.1	
1	#"Skriptdatei für den Testfall MS.H.2.1"
2	#"Anwendungsfall Motorsteuerung - Referenzpunktlauf (C-812 BoardID 4)"
3	#"19.07.2002"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	#"########## Testsequenz"
14	START,""
15	WAIT,5000
16	ACTION,MAIN,MENU,CLICK,"Einstellungen","Antriebe","Grundstellung..."
17	WINDOWEXISTS,"Grundstellung anfahren",YES
18	ACTION,"Grundstellung anfahren",COMBOBOX,SELECT,"Y","Antriebsauswahl"
19	ACTION,"Grundstellung anfahren",BUTTON,CLICK,"Absoluter Nullpunkt"
20	TEST,"Grundstellung anfahren",STATIC,TEXT,"Absolute Null für Motor Y","Meldungen"
21	ACTION,"Grundstellung anfahren",CHECKBOX,UNCHECK,"Position beibehalten"
22	ACTION,"Grundstellung anfahren",CHECKBOX,UNCHECK,"Grundstellung für alle Antriebe"
23	ACTION,"Grundstellung anfahren",CHECKBOX,CHECK,"Gültige Kalibrierungsdaten"
24	ACTION,"Grundstellung anfahren",BUTTON,CLICK,"Referenzpunktlauf"
25	TEST,"Grundstellung anfahren",BUTTON,DISABLED,"Absoluter Nullpunkt"

26	TEST,"Grundstellung anfahren",BUTTON,DISABLED,"Referenzpunktlauf"
27	TEST,"Grundstellung anfahren",COMBOBOX,DISABLED,"Antriebsauswahl"
28	TEST,"Grundstellung anfahren",STATIC,TEXT,"","Meldungen"
29	WAIT,5000
30	TEST,"Grundstellung anfahren",STATIC,TEXT,"Index Y","Meldungen"
31	WAIT,5000
32	TEST,"Grundstellung anfahren",STATIC,TEXT,"Target Pos Y","Meldungen"
33	HANDLEMESSAGEBOX,"Meldung",OK
34	TEST,"Grundstellung anfahren",BUTTON,ENABLED,"Absoluter Nullpunkt"
35	TEST,"Grundstellung anfahren",BUTTON,ENABLED,"Referenzpunktlauf"
36	TEST,"Grundstellung anfahren",COMBOBOX,ENABLED,"Antriebsauswahl"
37	ACTION,"Grundstellung anfahren",BUTTON,CLICK,"Ok"
38	WINDOWEXISTS,"Grundstellung anfahren",NO
39	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
40	WAIT,5000
41	#"########## Nachbereitung"
42	CLEANUP
43	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
44	DELETE,TGT,"XCONTROL.BAK"
45	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
46	DELETE,TGT,"HARDWARE.BAK",FORCE

Test_MS.H.3.1	
1	#"Skriptdatei für den Testfall MS.H.3.1"
2	#"Anwendungsfall Motorsteuerung - Optimieren (C-812 BoardID 4)"
3	#"20.12.2002"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	COPY,REF,"Y.BMP",TGT,"Ref.BMP"
14	#"########## Testsequenz"
15	START,""
16	WAIT,7000
17	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
18	WINDOWEXISTS,"Manuelle Justage",YES
19	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Y","Aktueller Antrieb"
20	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
21	WINDOWEXISTS,"Manuelle Justage",NO
22	ACTION,MAIN,MENU,CLICK,"Einstellungen","Antriebe","Optimieren..."
23	WINDOWEXISTS,"C812 DCM-Parameter für",YES
24	WINDOWEXISTS,"LineScan",YES
25	ACTION,"C812 DCM-Parameter für",EDITBOX,EDIT,"200","Distanz"
26	ACTION,"C812 DCM-Parameter für",BUTTON,CLICK,"Start Scan"
27	WAIT,2000
28	LAUNCH,BIN,"i_view32.exe","Ref.BMP",NOWAIT
29	QUESTION,"Entspricht die Bildschirmausgabe der dargestellten Grafik ?",YES
30	ACTION,"C812 DCM-Parameter für",BUTTON,CLICK,"Ok"
31	WINDOWEXISTS,"C812 DCM-Parameter für",NO
32	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
33	WAIT,5000

34	#"########## Nachbereitung"
35	CLEANUP
36	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
37	DELETE,TGT,"XCONTROL.BAK"
38	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
39	DELETE,TGT,"HARDWARE.BAK",FORCE
40	DELETE,TGT,"Ref.BMP",FORCE

Test_MJ.H.3.2	
1	#"Skriptdatei für den Testfall MS.H.3.2"
2	#"Anwendungsfall Motorsteuerung - Optimieren (C-832 BoardID 0)"
3	#"20.12.2002"
4	#"########## Vorbereitung"
5	EXISTS,TGT,"XCONTROL.INI"
6	EXISTS,TGT,"HARDWARE.INI"
7	EXISTS,TGT,"STANDARD.MAK"
8	EXISTS,TGT,"TESTDEV.DAT"
9	COPY,TGT,"XCONTROL.INI",TGT,"XCONTROL.BAK"
10	COPY,ENV,"TEST_XCONTROL.INI",TGT,"XCONTROL.INI"
11	COPY,TGT,"HARDWARE.INI",TGT,"HARDWARE.BAK"
12	COPY,ENV,"MJ_HARDWARE.INI",TGT,"HARDWARE.INI",FORCE
13	COPY,REF,"Theta.BMP",TGT,"Ref.BMP"
14	#"########## Testsequenz"
15	START,""
16	WAIT,7000
17	ACTION,MAIN,MENU,CLICK,"Ausführen","Manuelle Justage (Alt)..."
18	WINDOWEXISTS,"Manuelle Justage",YES
19	ACTION,"Manuelle Justage",COMBOBOX,SELECT,"Theta","Aktueller Antrieb"
20	ACTION,"Manuelle Justage",BUTTON,CLICK,"Verlassen"
21	WINDOWEXISTS,"Manuelle Justage",NO
22	ACTION,MAIN,MENU,CLICK,"Einstellungen","Antriebe","Optimieren..."
23	WINDOWEXISTS,"C832 DCM-Parameter für",YES
24	WINDOWEXISTS,"LineScan",YES
25	ACTION,"C832 DCM-Parameter für",EDITBOX,EDIT,"200","Distanz"
26	ACTION,"C832 DCM-Parameter für",BUTTON,CLICK,"Start Scan"
27	WAIT,2000
28	LAUNCH,BIN,"i_view32.exe","Ref.BMP",NOWAIT
29	QUESTION,"Entspricht die Bildschirmausgabe der dargestellten Grafik ?",YES
30	ACTION,"C832 DCM-Parameter für",BUTTON,CLICK,"Ok"
31	WINDOWEXISTS,"C832 DCM-Parameter für",NO
32	ACTION,MAIN,MENU,CLICK,"Datei","Beenden"
33	WAIT,5000
34	#"########## Nachbereitung"
35	CLEANUP
36	COPY,TGT,"XCONTROL.BAK",TGT,"XCONTROL.INI"
37	DELETE,TGT,"XCONTROL.BAK"
38	COPY,TGT,"HARDWARE.BAK",TGT,"HARDWARE.INI",FORCE
39	DELETE,TGT,"HARDWARE.BAK",FORCE
40	DELETE,TGT,"Ref.BMP",FORCE