Studienarbeit zum Thema:

Reverse-Engineering des Subsystems Detektoren des RTK-Steuerprogramms

Jan Picard
René Harder
Alexander Paschold

Berlin im November 2000

Gliederung der Studienarbeit

I. Zusammenfassen der Aufgabenstellung, Erläutern der Ziele der Studienarbeit und des Umfangs des vorliegenden Dokuments

II. Erläutern der Vorgehensweise (Ansätze, Werkzeuge)

III. Abgrenzung der Implementation der Detektoren
(Dateien, Klassen)

IV. Dokumentation des Ist-Zustandes
(Klassen, Typen, Variablen, Modularität, Mechanismen, **Fenster,** Fehler)
⇒ Referenz zu Klassenbeschreibungen...

V. Grafiken: Vererbungsbeziehungen (Klassen)

VI. Dokumentation und Bewertung des Interfaces

VII. Bemerkungen zur Portabilität

VIII. Pflichtenheft + Erläuterungen zum ini-File

IX. Dokumentation zum Einbinden neuer Detektoren

X. Bewertung des Ist-Zustandes & Gedanken zur Soll-Architektur

I. Einleitung

Zusammenfassung der Aufgabenstellung

Unsere Aufgabe war es, den Bereich der Implementation der Detektoren (0- und 1 dimensionale) vom restlichen Programm abzugrenzen und zu dokumentieren, um die Weiterentwicklung und Portierung auf andere Systeme zu ermöglichen. Dabei wurde auf die Dokumentation der Softwarearchitektur Wert gelegt, welche aus den Teilen IST- und SOLL-Zustand besteht, letztere wird dabei jedoch nur kurz angerissen in Form von Hinweisen auf zu behebende Design- und Implementationsfehler. Zum IST-Zustand gehört das Erfassen der vorgefundenen Struktur mit folgenden Fragen: Wo sind die zu den Detektoren gehörenden Teile definiert/implementiert ? Wurde die saubere Trennung der für die Detektoren benötigten Bestandteile vom restlichen Programm erreicht ? Wie sehen die Include-, Vererbungs- und Aufrufbeziehungen sowie Klassenhierachien aus? Wurde eine konsequente objektorientierte Implementation erreicht ? Ebenso gilt es zu klären, wie die Hierarchien der Softwareschichten beschaffen sind. Abschließend dazu ist aus dem IST-Zustand ein Pflichtenheft abzuleiten.

Kurzer Überblick über das Dokument

Das vorliegende Dokument wird zunächst erklären, wie wir bei der Analyse vorgegangen sind und welche Werkzeuge dabei von uns verwendet bzw. entwickelt worden sind. Darauf folgt die Darlegung der Erkenntnisse, die wir bezüglich der Abgrenzung gewannen, mit einem Überblick über die Dateien, die zu den Detektoren gehören und den darin implementierten Klassen und Funktionen mit den dazugehörigen Typen und Variablen.
Daran schließt sich die Dokumentation des IST-Zustandes an, in der beschrieben wird welche Detektoren zur Zeit unterstützt werden, welche Mechanismen damit in Zusammenhang stehen und wie diese arbeiten. Im Anschluß an diesen Teil werden wir grafisch die Vererbungsbeziehungen zeigen. Dokumentation und Bewertung des „Interfaces“ c_layer.h folgen diesem Abschnitt. An diesem Punkt werden wir Gedanken zur Portabilität des Programms äußern. Der achte Abschnitt beinhaltet die Pflichtenhefte 0- und 1-dimensionaler Detektoren, sowie eine Dokumentation des Detektorbereichs des ini-Files. Das Einbinden von neuen Detektoren wird im Anschluß erklärt. Zu guter letzt nehmen wir eine Bewertung des IST-Zustandes vor und machen uns Gedanken bezüglich des SOLL-Zustandes.

II. Erläutern der Vorgehensweise

Zu Beginn unserer Arbeit hatten wir ein Programm im Umfang von ca. 27.000 Zeilen im Quelltext vorliegen, ohne nennenswerte Kommentierung. Dokumente über das Programm lagen uns in Form eines Pflichtenheftes im Anfangsstadium und einer Sammlung von Hardwaredokumentationen bzw. Bedienungshandbüchern zu den Detektoren vor. Der erste Schritt bestand nun darin, den Bereich der 0- und 1-dimensionalen Detektoren eindeutig vom übrigen Programm abzugrenzen. Da die Bezeichnung der einzelnen Module uns als Kriterium für die Zuordnung zu unserem Gebiet zu wage erschien, bedurfte es einer speziellen Methode zur Ermittlung der Codeteile, die wir zu dokumentieren hatten. Die Verwendung des Bauhaus-Tools schied von vornherein aus, da die vorliegenden Quellen C++-Elemente enthalten, das Bauhaus-Projekt aber mit C++ nicht umgehen kann. Ausschließlich das Top-Down-Verfahren anzuwenden war ebenfalls nicht möglich aufgrund des nur rudimentär vorhandenen Pflichtenheftes. Das Buttom-Up-Verfahren schien uns am effektivsten zu sein. Im Gegensatz zum Top-Down-Verfahren, bei dem man von der Nutzeroberfläche, also dem Programm an sich und dem Pflichtenheft ausgeht und so versucht die Quellen zu dokumentieren, geht man beim Buttom-Up-Verfahren von den Quellen aus. Somit entschieden wir uns dafür, beide Verfahren miteinander zu kombinieren, und somit eine effektive Lösung des Problems in greifbare Nähe zu rücken.
Das gesamte Projekt entstand unter Borland C++ V4.52, somit gehörte diese Entwicklungsumgebung automatisch zu den von uns verwendeten Werkzeugen. Bei genauerer Betrachtung dieses Werkzeuges wird jedoch schnell klar, daß bei weitem nicht alle Informationen die man für ein solches Vorhaben benötigt von Borland C++ V4.52 geliefert werden können. Es diente uns vorwiegend zum kompilieren, debuggen und als Sprachreferenz. Als weiteres Werkzeug wählten wir SNIFF+ 3.2 [1)] aus. Dabei handelt es sich um eine sehr flexible, umfangreiche Entwicklungsumgebung, die neben C++ viele andere Programmiersprachen versteht. Es ermöglicht die Suche über das gesamte Projekt ebenso wie die grafische Darstellung der Includebeziehungen und der Klassenhierarchie, um nur einige der Features zu nennen.
Zu allererst wollten wir wissen, wann welche Funktion des Programms aufgerufen wird. Da dies jedoch zweifelsohne eine immense Flut an Informationen bedeutet, stand vorab bereits die Frage nach Möglichkeiten, den Überblick zu behalten und das für unsere Zwecke Wichtigste herauszufiltern. Dafür entsannen wir uns der Use-Case-Methode. Das bedeutet im Wesentlichen folgendes: man legt bestimmte Abläufe der Programmbedienung fest, die sogenannten Use-Cases, und protokolliert dann bei exakter Abarbeitung eines solchen Use-Cases die programmseitig ablaufenden Geschehnisse. So weit, so gut. Was, warum und wie waren weitestgehend geklärt, nur das womit bereitete Schwierigkeiten, im Klartext: Womit kommt man an die Funktionsaufrufe zu einem bestimmten Use-Case ? Die Untersuchung der uns bekannten Tools brachte keine optimale Lösung. Somit hatten wir die Idee, daß uns jede implementierte Funktion selbständig mitteilt, wann sie aufgerufen wird. Dies war am einfachsten dadurch zu realisieren, daß sämtliche Funktionsrümpfe am Anfang durch ein fprintf() ergänzt wurden, welches in eine bestimmte Datei einen Zeitstempel, den Namen der Funktion und des Moduls schreibt. Diese Methode hat jedoch Nachteile. Zum einen handelt es sich offensichtlich um eine Quelltextmodifikation, was aber nicht weiter tragisch ist, da sie nur temporär ist und kontrolliert erfolgt, und zum anderen wird sehr schnell klar, daß das Einfügen der fprintf() Anweisungen per Hand in die Funktionen einen unverhältnismäßigen Aufwand bedeutet. Dieses Problem forderte eine Lösung in Form eines Programms, welches diese Arbeit für jedes Modul ausführt.

[1)] http://www.takefive.com

Die selbstentwickelten Werkzeuge

So entstand das Tool tracer.exe . Man übergibt ihm die Datei, die modifiziert werden soll und die Ausgabedatei. tracer.exe sucht nun nach Funktionsköpfen und fügt direkt hinter einem solchen eine fprintf()-Anweisung ein. Die von tracer.exe getätigten Änderungen in den Quellfiles werden wie folgt markiert:

```
//hpp************************************************************
Anweisungen...
//pph************************************************************
```

Die Anweisung, die jeder Funktion hinzugefügt wurde, sieht folgendermaßen aus:

```
fprintf(trace,“%i    DATEINAME   FUNKTIONSNAME\n“,time(zidx));
```

Da dieses Tool noch nicht zur Vollkommenheit erwachsen ist, bedarf es bei der nun modifizierten Datei noch etwas Nacharbeit. Und zwar ist die Implementation des TRACE momentan noch manuell zu vervollständigen.

Dabei muß jede Quelltextdatei um den Eintrag

```
#include <time.h>,
```

die Hauptfiles der Module (global) mit

```
FILE *trace;
time_t *zidx;
```

ergänzt werden. In den Hauptfiles der Module muß in den Einsprungfunktionen *LibMain()* und *WinMain()*

```
trace = fopen(TRACEFILENAME,“w“);
setbuf(trace,(char*)NULL);
```

hinzugefügt werden. Allen anderen Dateien wird (global)

```
extern FILE *trace;
extern time_t *zidx;
```

dazu gefügt.
Nach diesen Modifikationen der .cpp-Dateien ist das gesamte Projekt neu zu übersetzen. War dies erfolgreich, so liegt eine Version von Develop.exe vor, die sogenannte Tracefiles erzeugt. Hat der Compiler an den Dateien etwas auszusetzen - dies kann bei bestimmten if , switch oder ähnlichen Strukturen der Fall sein, wenn tracer.exe diese als Funktionsrümpfe erkannt hat - ist nur das fälschlich eingefügte fprintf() zu entfernen. Die von Develop.exe erzeugten Tracefiles - vier an der Zahl aufgrund der vier separaten Programmodule - heißen wie folgt:

trace.spl
trace.mtr
trace.ctr
trace.dev

Anhand der Endungen lassen sich die Tracefiles den Modulen zuordnen:

.spl	⇒	SpLib.dll mit Hauptfile: *l_layer.cpp*
.mtr	⇒	Motors.dll mit Hauptfile: *m_layer.cpp*
.ctr	⇒	Counters.dll mit Hauptfile: *c_layer.cpp*
.dev	⇒	Develop.exe mit Hauptfile: *m_main.cpp*

Der Inhalt dieser Dateien hat folgende Bedeutung.

Zeitindex Sourcefile Funktionsname

Der Zeitindex hat z.Z. eine Auflösung von einer Sekunde, was zugegebenermaßen nicht optimal ist, eine bessere Variante war rahmensprengend und somit verwendeten wir die einfachere Windowsfunktion *time()*. Wie gesagt, tracer.exe ist nicht vollkommen, aber für uns in dieser Form ausreichend und deswegen solchen Einschränkungen unterworfen.
Um nun maximalen Nutzen aus den Tracefiles zu ziehen, haben wir Use-Cases festgelegt und simuliert. Dies waren bei uns im wesentlichen

Programm starten und unverzüglich beenden
Messung starten/stoppen (Zählerfenster)
Detektor konfigurieren

Die bei diesen Use-Cases erzeugten Tracefiles haben wir gesichert und miteinander verglichen. Dabei war festzustellen, daß die Aufbereitung dieser Daten recht umfangreich ist, was die Entwicklung weiterer Tools nach sich zog. Neben tracer.exe entstanden essenz.exe, mehrfach.exe, vergleich.exe und zyklen.exe.
essenz.exe: liefert eine Liste der Funktionen, welche in einem Tracefile stehen, d.h. jede Funktion wird nur einmal in dieser Liste erscheinen.
mehrfach.exe: dient dazu aufeinanderfolgende identische Einträge eines Tracefiles zu zählen und auf einen Eintrag zu reduzieren. Die Anzahl der Wiederholungen wird hinter den Eintrag geschrieben.
vergleich.exe zeigt die Unterschiede zweier Tracefiles auf, indem es diese in eine Ausgabedatei schreibt.
zyklen.exe: versucht in einer Abfolge von Einträgen einen Zyklus zu finden, ist ein solcher gefunden worden, wird gezählt wie oft ein solcher Zyklus hintereinander vorkommt, die Wiederholungen werden entfernt und die Anzahl dazu geschrieben.
Soviel zu den verwendeten Werkzeugen. Anhand der so gewonnenen Daten und der recht mageren, aber teilweise doch vorhandenen Kommentierung konnten wir den Bereich der Detektoren immer mehr eingrenzen. Das Ergebnis dieser Arbeit führte zu folgendem Abschnitt.

III. Abgrenzung der Implementation der Detektoren

Nach umfangreichen Analysen der Quellen mit den im vorherigen Abschnitt beschriebenen Mitteln sind unserer Meinung nach die folgenden Dateien ganz oder teilweise dem Bereich der 0- bzw. 1-dimensionalen Detektoren zuzuordnen. Der Hauptteil der Implementation der Detektoren befindet sich in dem Modul *counters.dll* :

Dateien

- am9513a.h
  Deklaration der Klasse TAm9513a, Deklaration von Konstanten und Datentypen

- c_layer.h
  Interface zu c_layer.cpp (Interface zu *counters.dll*)

- comhead.h
  Deklaration von Datentypen (TDeviceType, THowReadOutPsd, TPsdDataType, THardware, TUnitType) und Returncodes

- dfkisl.h
  Deklaration von Konstanten für den *Radicon SCSCS* (kisl1.c)

- dlg_tpl.h
  Deklaration von allgemeinen Dialogfensterklassen (TModalDlg, TModelessDlg)

- m_devcom.h
  Deklaration der Klassen TDevice & TDList, Deklaration des Datentyps TDSettings

- m_devhw.h
  Deklaration der Klassen TGenericDevice, TEncoder, TRadicon , TCommonDevParam & TScsParameters

- m_psd.h
  Deklaration der Klassen TPsd, TStoe_Psd, TBraun_Psd & TPsdParameters und zugehöriger Datentypen

- prkmpt1.h
  Interface zu kmpt1.c (low-level-Treiber für *Radicon SCSCS*)

- radicon.h
  Interface zu kisl1.c (mid-level-Treiber für *Radicon SCSCS*)

- rc_def.h
  Definition von Nachrichten (#define)

- am9513a.cpp
  Implementation der Klasse TAm9513a (Interface Controllerkarte *AXIOM AX5216* (*Am9513a*))

- braun_psd.cpp
  Implementation der Klasse TBraun_Psd (Interface Detektor *BraunPSD*)

- c_layer.cpp
  Implementation des Interfaces zu *counters.dll*, implementiert einige Methoden der Klasse TDList

- counters.cpp
  Implementation der Klassen TDList (Verwaltungsstruktur für Detektoren), TDevice (Wurzelklasse für alle Detektoren, Testdetektor), TRadicon (Interface Detektor *Radicon SCSCS*), TPsd (Interface für Detektoren vom Typ *PSD*), TStoe_Psd (Interface Detektor *Stoe PSD*), TGenericDevice (Interface für Detektoren an Contollerkarte *AXIOM 5216*), TEncoder, TCommonDevParam (Dialog *Zähler-Konfiguration*), TScsParameters (Dialog *Settings for Radicon SCS*)

- dlg_tpl.cpp
  Implementation der Klassen TModalDlg & TModelessDlg

- kisl1.c
  Implementation von Funktionen für den Hardwarezugriff auf den *Radicon SCSCS* (mid-level)

- kmpt1.c
  Implementation von Funktionen für den Hardwarezugriff auf den *Radicon SCSCS* (low-level)

- m_dlg.cpp (gehört nicht zum Modul *counters.dll*)
  Implementation der Klasse TPsdParameters (Dialog *Einstellungen für den PSD*)

- counters.def
  Moduldefinitionsdatei für *counters.dll* (Borland C++ 4.5)

- counters.rc
  Ressourcenvorlage der Dialogfenster *Einstellungen für SCS* & *Zähler-Konfiguration*

- main.rc (gehört nicht zum Modul *counters.dll*)
  Ressourcenvorlage des Dialogfensters *Einstellungen für den PSD*

Nachfolgend sind sämtliche den Detektoren zuzuordnende Klassen aufgelistet.

Klassen

- TDList
  Implementiert eine Verwaltungsstruktur für die Detektoren.

- TAm9513a
  Kapselt den Zugriff auf die Controllerkarte AXIOM AX5216 mit dem Controllerchip Am9513a.

- TDevice
  Wurzelklasse für alle Detektorklassen und gleichzeitig auch das Interface zu allen Detektoren. Sie definiert grundlegende Methoden bzw. implementiert diese teilweise (Testdetektor).

- TGenericDevice
  Implementiert die Methoden für einen generischen Detektor, also für einen Detektor, der an der AXIOM AX5216 Schnittstellenkarte betrieben wird (z.B. russischer SCSCS).

- TRadicon
  Implementiert den Zugriff auf den Radicon-Szintilisationszähler. Wichtigste Aufgabe der Klasse ist die Kapselung des Zugriffs auf die Hardware mittels der low- und mid-Level-Treiber, die für den Radicon-Zähler in C implementiert sind

- TPsd
  Implementiert das Interface für die eindimensionalen Detektoren vom Typ P(ositional) S(ensitive) D(evice).

- TBraun_Psd
  Kapselt den Zugriff auf die Hardware des Detektors *BraunPSD*.

- TStoe_Psd
  Kapselt den Zugriff auf die Hardware des Detektors *StoePSD*.

- TEncoder
  Implementiert die Geräteklasse *Encoder*.

- TCommonDevParam
  Implementiert einen Dialog zur Einstellung der allgemeinen Parameter der Detektoren. (*Zähler-Konfiguration*)

- TScsParameters
  Implementiert einen Dialog zur Einstellung der Parameter des 0-dimensionalen Detektors *Radicon SCSCS*. (*Settings for Radicon SCS*)

- TPsdParameters
  Implementiert einen Dialog, in dem gerätespezifische Einstellungen für einen Psd gemacht werden können. Die Klasse bietet somit die grafische Nutzerschnittstelle zu den Psd-Geräteeinstellungen. (*Einstellungen für den PSD)*

IV. Dokumentation des IST - Zustandes

Modularität

Das Modul *counters.dll*, welches den Hauptbestandteil der Implementation der Detektoren bildet, ist in zweierlei Hinsicht nicht vollständig abgeschlossen.

(1) Die zur Implementation der Detektoren gehörige Klasse *TPsdParameters* ist nicht im Modul counters.dll enthalten, sondern wird in der zum Modul *Develop.exe* gehörigen Datei *m_dlg.cpp* implementiert. Die diesem Dialogfenster entsprechende Ressourcenvorlage befindet sich zwangsweise ebenfalls im Modul *Develop.exe* in der Datei *main.rc .*

(2) Innerhalb des Moduls *counters.dll* werden einige Symbole benötigt, die in anderen Modulen definiert sind. Das sind im einzelnen:

l_layer.cpp:	GetCFile(), SetInfo(), CreateDefaults(), SetStatus(), UnitEnum(), Delay(), DelayTime(), GetClientHandle(), GetFrameHandle(), GetMainInstance()
m_layer.cpp:	mGetDistance(), mlGetDistance(), mlGetIdByName(), mlIsAxisValid()
m_curve.cpp:	TCurve::New(), TCurve::FastOpen(), TCurve::FastClose(), TCurve::FastPAdd()
ctl3d.h:	Ctl3dColorChange()

Ohne Definition dieser Symbole läßt sich das Modul *counters.dll* nicht eigenständig übersetzen. Abhilfe kann hier durch selbstdefinierte Stubs für diese Symbole geschaffen werden, wobei allerdings die Funktionalität leiden würde (wurde erfolgreich getestet).

Fazit:
Man kann definitiv nicht davon sprechen, daß die Detektoren in einem eigenständigen Modul implementiert sind.

Typen und Strukturen

Comhead.h

TDeviceType

Beschreibung: Dieser Typ dient der Einteilung der Geräte in Geräteklassen.

Definition:

```
typedef enum
{
      CounterDevice = 1,
      PsdDevice,
      MonitorDevice,
      EncoderDevice,
      GenericDevice
} TDeviceType;
```

THardware

Beschreibung: Dieser Typ dient zur Identifizierung des Hardwaretyps.

Definition:

```
typedef enum
{
      NothingHW = 2000,
      RadiconHW,
      BraunHW,
      GenericPsdHW,
      Am9513HW,
      StoeHW,
      MatroxHW,
      EncoderHW
} THardware;
```

THowReadOutPsd

Beschreibung: Dieser Typ enthält vier Werte, die der Festlegung der Lesemethode beim Auslesen von PSD-Devices dienen.

Definition:

```
typedef enum
{
      FinalRead = 1,          // abschließendes Lesen
      FirstRead,              // eröffnendes Lesen
      AccumulationRead,       // akkumulierendes Lesen
      IntermediateRead        // vermittelndes Lesen
} THowReadOutPsd;
```

TPsdDataType

Beschreibung: Dieser Typ wird bei der Festlegung des Typs der Daten vom Braun_PSD benötigt (SetDataType())

Definition:

```
typedef enum
{
      PsdEnergyData = 3000,
      PsdPositionData
} TPsdDataType;
```

TScaleType

Beschreibung: Bei der Festlegung, um welchen Skalentyp es sich handelt, wird dieser Typ benötigt.

Definition:

```
typedef enum
{
      Logarithmic = 1,  //logarithmische Skala
      Linear,           //lineare Skala
      User              //benutzerspezifizierte Skala
} TScaleType;
```

TUnitType

Beschreibung: Hierbei handelt es sich um die Aufzählung von Einheiten für Winkel- und Längenmessungen. Auch die Einheit Kanal und <nichts> sind in diesem Typ festgelegt.

Definition:

```
typedef enum
{
      Grad = 1,
      Minuten,
      Sekunden,
      Millimeter,
      Mikrometer,
      Channel,
      None
} TUnitType;
```

M_devcom.h

TDSettings

Beschreibung: In dieser Struktur kann die Konfiguration eines Detektors abgespeichert werden.

Definition:

```
typedef struct
{
      WPARAM wpJumpTo;          //wird zum Übergeben von
                                //Messageparametern verwendet
      float  fExposureTime;     //Meßlimit Zeit
      DWORD  dwExposureCounts;  //Meßlimit Impulse
      float  fFailure;          //bestimmter Wert des Meßfehlers
      int    nFunctionId;       //aktueller Betriebsmodus bzw. Zustand des
                                //Gerätes, mögl. Werte stehen in
                                //m_devcom.h als #define...
      BOOL   bStaticFailure;    //Meßfehler auf bestimmtem Wert halten
      int    nAddedChannels;    //Anzahl der zusammengefaßten Kanäle
} TDSettings;
```

M_psd.h

MessDatenHeader

Beschreibung: Diese Struktur wird benötigt, um den Header für das Datenfile zu konstruieren.

Definition:

```
struct   MessDatenHeader
{
      long        HighADCCounts;
      long        LowADCCounts;
      long        HighEGYCounts;
      long        LowEGYCounts;
      long        HighPOSCounts;
      long        LowPOSCounts;
      long        HSumDetCounts;
      long        LSumDetCounts;
      long        RateADC;
      long        RateEGY;
      long        RatePOS;
      long        RateDET;
      long        Messzeit;
      long        HVParameter;
      long        EndLifeTime;
      long        Ueberlauf;
} Header;
```

Globale Variablen und Konstanten in den .cpp Modulen

In diesem Abschnitt werden die globalen Variablen und Konstanten jener .cpp-Module erläutert, die wir in ihrer Funktionalität den Detektoren zugeordnet haben.

Am9513a.cpp

In *Am9513a.cpp* sind keine globalen Variablen oder Konstanten definiert.

Braun_PSD.cpp

static HINSTANCE AsaDllInstance

Diese Variable dient dazu, das Handle zum Modul *asa.dll* aufzunehmen.

static char buf

Hier handelt es sich um eine Puffervariable die für Verschiedenes genutzt wird. Teilweise sind in den Funktionen eigene, lokale *buf* – Variablen definiert.

extern LPDList lpDList

Hierbei handelt es sich um einen Zeiger auf das TDList - Objekt.

const char TBraunError*

Hierbei handelt es sich um ein Feld von Stringkonstanten, die bei der Ausgabe von Fehlermeldungen mittels *MessageBox()* in *Braun_PSD.cpp* verwendet werden.

BYTE TBraun_Psd::befehl

Hier wird das in der Klasse *TBraun_PSD* definierte Feld mit Werten versehen, welche vermutlich bestimmte Befehle darstellen, die auch teilweise kommentiert sind. Diese Befehle werden an *BuildOperation()* übergeben.

BYTE TBraun_Psd::echo

Dieses Feld ist ebenfalls in der Klasse *TBraun_PSD* definiert worden und wird hier mit Werten versehen. Im weiteren werden die Elemente dieses Feldes genauso behandelt wie die des Feldes *befehl*, *echo* wird lediglich bei *BuildOperation()* nicht als erstes Argument übergeben, sondern als zweites. Die Anfangs-Initialisierung unterscheidet sich von *befehl,* jedoch nicht völlig, d.h. einige Werte sind identisch. Die Bedeutung ist nicht völlig klar, es scheint sich jedoch um eine Art Referenzwert bezüglich der Komunikation mit dem *Braun_PSD* zu handeln.

C_layer.cpp

LPSTR pTime

Diese Variable ist ein Zeiger (im Falle von WIN16) auf den BIOS-Speicherbereich für die Systemzeit. Im Falle WIN32 wird sie lediglich definiert. *crnt_time()* liefert *pTime* zurück.

HINSTANCE hModuleInstance

Hierbei handelt es sich um die Variable, welche die Instanz der DLL identifiziert, sie wird durch *LibMain()* gesetzt und entspricht deren ersten Parameter.

static char dlVersion

Diese Variable enthält einen Versionsstring (momentan V 1.52 und das Datum der Datei via Makro __DATE__). Mittels *dlGetVersion()* wird die Variable zurückgegeben, es ist der einzige Zugriff.

const int nMaxDeviceAllowed

Hiermit wird die Anzahl der maximal erlaubten angeschlossenen Geräte festgelegt. Dieser Wert wird bei der Erzeugung des *TDList* Objektes übergeben und ist momentan auf 3 gesetzt, dadurch werden nur die ersten drei Geräte die in der ini-Datei angegeben sind in die Liste aufgenommen und initialisiert, darüber hinaus in der Datei angegebene Geräte werden ignoriert. (siehe Dokumentation des ini-File)

static int bModulLoaded

Diese Variable dient als Statusflag dafür, ob das Modul geladen ist oder nicht. Sie wird in *InitializeCountersDLL()* auf TRUE gesetzt, dann wird *InitializeModule()* des Objektes ausgeführt.

extern LPDList lpDList

Hierbei handelt es sich um einen Zeiger auf die Detektoren-Verwaltungliste. Die Variable wird hier im Abstand von fünf Zeilen zwei mal definiert.

static HINSTANCE hMotorDll

Diese Variable wird verwendet, um das Handle von *motors.dll*, ermittelt durch *GetModuleHandle(),* aufzunehmen. Der Wert wird jedoch nie wieder benötigt, also auf die Variable wird nie wieder zugegriffen.

static int retval

dient der Verarbeitung von ReturnValues, siehe *retval* unter Counters.cpp

int Failure

Die Bedeutung dieser Variablen ist unbekannt, nach ihrer Definition kommt sie im gesamten Projekt nicht mehr vor.

int FakeDevice

Diese Variable wird bei ihrer Definition auf 0 gesetzt, sonst erfolgt nie wieder ein Schreibzugriff. In *Kisl1.c* wird ihr Inhalt fünf mal auf *TRUE* getestet, dieser Wert kann nie enthalten sein, somit ist der Zweck dieser Variable nicht zu klären, ebenso können die fünf Abfragen als toter Code erklärt werden.

int FakeData

Diese Variable wird nur definiert, sie taucht im gesamten Projekt nie wieder auf.

static int rdd,rcc

Bei diesen Variablen handelt es sich vermutlich um eine lokale Variante der SCS Daten (*Rdd*) und Status (*Rcc*) Register, jedoch werden sie nie mit gültigen Daten beschrieben, verwendet aber schon und zwar in *getinf()* in der Funktion *InquireIntensity_SCS()*, welche allerdings nie aufgerufen wird.

static HWND hControlWnd

Diese Variable dient dazu ein Fensterhandle aufzunehmen.

static float Intensity_SCS

Die Variable wird nur definiert, und in *GetIntensity_SCS()* zurückgegeben. Sie erhält nie einen Wert zugewiesen und wird nie ausgewertet. Der Sinn ist somit nicht nachvollziehbar.

static DWORD expcounts

Auch diese Variable bekommt nie einen Wert zugewiesen. In *InquireIntensity_SCS()* wird sie bei einer Operation verwendet, sonst nie wieder.
Es scheint sich bei *InquireIntensity_SCS()* um eine Funktion in der Entwicklung zu handeln, da viele der verwendeten Variablen nie einen Wert zugewiesen bekommen.

static UINT nEvent

Hier wird der Identifizierungscode von TimerEvents gespeichert.

static int Cycle

Hierbei scheint es sich um eine Art Laufvariable zu handeln. Sie wird in *InitializeTDC_Event()* auf 100 gesetzt und in *InquireIntensity_TDC()* dekrementiert und auf 0 getestet. Beide Funktionen werden nie aufgerufen.

static float Intensity_TDC

Diese Variable wird (vermutlich) bei Intensitätsberechnungen in *InquireIntensity_TDC()* verwendet. *GetIntensity_TDC()* liefert sie zurück. Beide Funktionen werden nie aufgerufen.

static float Intensity_A913

Diese Variable wird in *InquireIntensity_A913()* auf 1.0 gesetzt und in *GetIntensity_A913()* zurückgeliefert, mehr wird mit *Intensity_A913* nicht gemacht.
Auch bei *InquireIntensity_A913*() scheint es sich um eine nicht beendete Implementation zu handeln.

Counters.cpp

IoWaitStoe

Dieser Wert stellt eine Verzögerungskonstante für Hardwarezugriffe im Zusammenhang mit dem Stoe_PSD dar.

const int Tx

Die Bedeutung dieser Konstanten ist völlig unklar, sie wird im gesamten Projekt nie wieder genutzt.

LPDList lpDList

Hierbei handelt es sich um einen Zeiger auf das *TDList* - Objekt.

static int nEventCalls

Diese Variable scheint eine bestimmte Anzahl von Ereignissen zu speichern, wenn diese erreicht ist, wird eine Message *SteeringReady* abgeschickt.

static int nEvent

Hier wird die Identifizierungskennung von Timerevents gespeichert.

static int nCalledEvents

Laufvariable, in welcher die eingetroffenen Events mitgezählt werden. *EventHandler*() nimmt diese Inkrementierung vor. *InitializeHander()* initialisiert sie.

static float fEventIntensity

Mit dieser Variable werden verschiedene Rechnungen ausgeführt, die Auswertung beschränkt sich jedoch auf zwei Stellen im Programm. Abhängig vom Inhalt werden *fEventSigma* entweder 1.0 zugewiesen, oder das Ergebnis einer Rechnung. (siehe *fEventSigma*) Daher ist die Bedeutung dieser Variable unklar.

static float fEventSigma

Die Bedeutung dieser Variable ist unklar, sie nimmt das Ergebnis einer mathematischen Operation mit *fEventIntensity* auf, bzw. wird auf 1.0 gesetzt, sie wird jedoch nie ausgewertet.

Unseres Erachtens nach sind diese beiden Variablen (*fEventSigma, fEventIntensity*) ohne Bedeutung, ebenso die dazugehörigen Strukturen. Sie stellen vermutlich eine Stufe in der Entwicklung des Programms dar.

static HWND hEventControlWnd

Diese Variable dient ausschließlich bei *PostMessage()* Aufrufen dazu das Empfängerfenster zu spezifizieren. Nach Borland Hilfe gibt es dafür zwei besondere Werte einer davon ist *NULL* und genau dieser wird jedesmal übermittelt an die Funktion, denn *hEventControlWnd* wird nie explizit gesetzt, nur einmal implizit bei ihrer Definition.

static BOOL bEventDataValid

Auch diese Variable ist offensichtlich ohne Funktion, sie wird nach ihrer Definition lediglich zweimal beschrieben (in Methode *EventHandler()*) und nie ausgewertet.

static BOOL bEventDeviceActive

Die gleiche Situation ergibt sich bei dieser Variable: sie hat keine Funktion, sie wird nur einmal beschrieben, ebenfalls nie ausgewertet.

static int MaxHighVoltage

Diese Variable wird nur bei ihrer Definition mit dem Wert 900 beschrieben, sonst nur ausgelesen (zwei mal). Sie scheint eine obere Grenze für die Hochspannung darzustellen.

static int retval

Die Aufgabe dieser Variablen besteht offensichtlich darin Rückkehrwerte aufzunehmen (RETurnVALue), es werden dabei die in *Comhead.h* definierten Rückkehrcodes übergeben und ausgewertet. Bei der Untersuchung dieser Variablen ist uns aufgefallen, daß viele der in Counters.cpp implementierten Funktionen *retval* lokal definiert haben, jedoch nicht alle.

Dlg_tpl.cpp

TModalDlg TheDialog; TModelessDlg* TheModeless*

Bei diesen Variablen handelt es sich um Zeiger auf ein *TModalDlg* bzw. *TModelessDlg* Objekt sie werden für den Zugriff auf die Methoden der entsprechenden Klassen verwendet.

extern HINSTANCE hModuleInstance

Hiermit wird die Variable, welche in *C_layer.cpp* definiert wurde bekannt gemacht. Hierbei handelt es sich um die Variable, welche die Instanz der DLL identifiziert, sie wird durch *LibMain()* in *C_layer.cpp* gesetzt und entspricht deren ersten Parameter.

M_dlg.cpp

TCScanParam CScan

Die Bedeutung dieser Variablen ist völlig unklar, da sie außer bei ihrer Definition sonst nie wieder im Programm auftaucht.

extern TMain Main; extern TSteering Steering

Hiermit werden die Objekte *Main* und *Steering* im Modul bekannt gemacht.

extern LPDList lpDList

Hierbei handelt es sich um einen Zeiger auf das *TDList* - Objekt.

extern BOOL bManualMovesCorrected

Diese Variable wird in *M_main.cpp* definiert und sogleich auf *FALSE* gesetzt. In *TAngleControl::Dlg_OnCommand()* erfolgt die einzige Auswertung. Neben ihrer Initialisierung wird sie nur noch durch den Key *Correction* in der Sektion *ManualMoves* der INI Datei beeinflußt (Auswertung dieses Keys und Übergeben des entsprechenden Wertes erfolgt im Konstruktor der Klasse *TMain*). Die genaue Bedeutung des Wertes ist unbekannt.

Klassenbeschreibungen

Auf den folgenden Seiten werden die zur Implementation der Detektoren gehörigen Klassen beschrieben. Folgendes wird dargestellt:

- UML Klassendiagramm
- Klassenhierarchie
- Friends
- Files – zu der Klasse gehörige Dateien
- Verantwortlichkeiten der Klasse
- Beschreibung der Klassenattribute
- Beschreibung der Klassenmethoden
- Fehler und Probleme

Bei der Beschreibung der Attribute und Methoden wird jeweils auf die Methoden bzw. Funktionen verwiesen, die auf selbige zugreifen. Dabei sind subsystemfremde Methoden bzw. Funktionen ***fett*** dargestellt.

class TDList

1. UML-Klassendiagramm

class TRadicon	
	- aDevice : TDevice**
	- hDInstance : HINSTANCE
	- MD_Counter : int
	- MD_Encoder : int
	- MD_Generic : int
	- MD_Monitor : int
	- MD_Psd : int
	- nActiveDevice : int
	- nLastDevice : int
	- nMaxNumber : int
	+ ~TDList(void)
	+ CalibrateDlg(TCurve&)
	+ DeleteDevices(void) : BOOL
	+ DP(int) : TDevice*
	+ DP(void) : TDevice*
<<inline>>	+ GetDevice(void) : int
	+ GetIdByName(TDeviceType) : int
<<inline>>	+ GetModuleHandle(void) : HINSTANCE
	+ InitializeModule(void) : BOOL
	+ IsDeviceValid(TDeviceType) : BOOL
	+ IsValidId(int id) : BOOL
<<inline>>	+ LastId(void) : int
	+ ParsingAxis(LPSTR) : TDeviceType
	+ SaveModuleSettings(void) : BOOL
	+ SetDevice(int) : BOOL
	+ SetParametersDlg(void)
	+ TDList(int, HINSTANCE)

2. Klassenhierarchie

TDList ist von keiner Klasse abgeleitet.

3. Friends

keine

4. Files

Klassendefinition: m_devcom.h
Implementation: counters.cpp
c_layer.cpp

5. Verantwortlichkeiten der Klasse

TDList verwaltet die angeschlossenen Detektoren in einem Array. Ein Detektor ist als der aktive festgelegt. Alle Subsysteme, welche die Detektoren nutzen, greifen im wesentlichen über diesen Mechanismus auf die Detektoren zu.

6. Beschreibung der Attribute

aDevice
Ist ein Array von Zeigern auf *Tdevice* - Objekten.
TDList::TDList(), *TDList::~TDList()*, *TDList::InitializeModule()*, *TDList::DP()*

hDInstance
Beinhaltet das Instanzen-Handle der *counters.dll*.
TDList::GetModuleHandle(), *TDList::TDList()*

MD_Counter, MD_Encoder, MD_Generic, MD_Monitor, MD_Psd
MD_xxx beinhaltet den Index des xxx-Gerätes im *aDevice*-Array.
TDList::TDList(), *TDList::InitializeModule()*, *TDList::GetIdByName()*, *TDList::IsDeviceValid()*

nActiveDevice
Ist der Index des aktiven Gerätes im *aDevice*-Array.
TDList::GetDevice(), *TDList::TDList()*, *TDList::InitializeModule()*, *TDList::DP()*
TDList::SetDevice()

nLastDevice
Ist der höchste Index eines tatsächlich vorhandenen Gerätes im *aDevice*-Array.
TDList::IsValidId(), *TDList::LastId()*, *TDList::TDList()*, *TDList::~TDList()*,
TDList::InitializeModule(), *TDList::DP()*, *TDList::SetDevice()*

nMaxNumber
Ist die maximale Anzahl möglicher Zähler im *aDevice*-Array.
TDList::TDList(), *TDList::InitializeModule()*

7. Beschreibung der Methoden

TDList
Initialisiert die Index-Variablen für die einzelnen Geräte mit –1 und legt das Array von Zeigern auf *TDevice*-Objekte an.
c_layer.cpp::LibMain()

~TDList
Zerstört die einzelnen *TDevice*-Zeiger und anschließend das Array von Zeigern auf *TDevice*-Objekte.

m_main.cpp:: FrameWndProc()

CalibrateDlg

CalibrateDlg wird nicht aufgerufen - scheint demnach keine relevante Funktion zu besitzen.

DeleteDevices

Gibt TRUE zurück.
m_main.cpp:: FrameWndProc*()*

DP(int)

Gibt einen Zeiger auf das Gerät zurück, das im Gerätearray den als Parameter gegebenen Index hat.
TDList::InitializeModule(), *TScsParameters::TScsParameters()*, *TCommonDevParam::TCommonDevParam()*, *TCommonDevParam::Dlg_OnInit()*, *TCommonDevParam::Dlg_OnCommand()*, ***TAreaScanParameters::TAreaScanParameters()***, ***TAreaScanParameters::SetDevice()***, ***TPsdRemoteSync::TPsdRemoteSync()***, ***TCalibratePsd::TCalibratePsd()***, ***TSetupAreaScan::Dlg_OnInit()***, ***TSetupAreaScan::Dlg_OnCommand()***, ***TPsdParameters::Dlg_OnInit()***, ***TSetupContinuousScan::Dlg_OnInit()***, ***TSetupContinuousScan::Dlg_OnCommand()***, ***TSetupScanCmd::TSetupScanCmd()***, ***TSteering::StartUp()***, ***TCounterWindow::TCounterWindow()***, ***TScanParameters::TScanParameters()***, ***TSetupStepScan::Dlg_OnInit()***, ***TSetupStepScan::Dlg_OnCommand()***, ***TTopographySetParam::Dlg_OnInit()***, ***TTopographySetParam::Dlg_OnCommand()***, *c_layer.cpp::FrameWndProc()*

DP(void)

Gibt einen Zeiger auf das aktive Gerät zurück.
TCommonDevParam::Dlg_OnCommand(), ***TCounterWindow::CanOpen()***, ***TCounterWindow::Create()***, ***TAngleControl::Dlg_OnInit()***, ***TScanParameters::TScanParameters()***, ***TScan::InitializeTask()***, ***TScan::CounterSetRequest()***, ***TSetupStepScan::Dlg_OnCommand()***, ***TMacroExecute::Dlg_OnCommand()***, ***TAdjustmentExecute::TAdjustmentExecute()***, ***TAdjustmentWindow::CounterSetRequest()***, ***TTopographyExecute::TTopographyExecute()***, ***TTopographySetParam::TTopographySetParam()***, *c_layer.cpp::dMeasureStart()*, *c_layer.cpp::dMeasureStop()*, *c_layer.cpp::dSetExposureValues()*, *c_layer.cpp::dGetExposureValues()*, *c_layer.cpp::InquireIntensity_SCS()*

GetDevice

Gibt den Index des aktiven Gerätes zurück.
TDList::SetParametersDlg(), c_layer.cpp::dlGetDevice()*,*
m_main.cpp::DoCommandsFrame()

GetIdByName

Gibt den Index des Gerätes zurück, das als Parameter gegeben ist.
TPsdParameters::Dlg_OnInit(), *c_layer.cpp::dlGetIdByName()*

GetModuleHandle

Gibt *hDInstance* zurück und wird nie aufgerufen.

InitializeModule

Liest aus dem ini-File die Gerätekonfiguration und erzeugt dementsprechend Zeiger auf *TDevice*-Objekte, die es in das Geräte-Array einfügt. Anschließend wird für jedes Gerät seine *Initialize()*-Methode aufgerufen, und so getestet, ob das Gerät angeschlossen ist und funktioniert. Sollte das bei einem Gerät nicht der Fall sein, so wird das *TDevice*-Objekt zerstört und aus dem Geräte-Array entfernt. Nun werden die Indexvariablen *MD_xxx* für die einzelnen Gerätetypen auf die entsprechenden Indizes gesetzt.
c_layer.cpp::InitializeCountersDLL()

IsDeviceValid

Überprüft, ob die zum Gerätetyp gehörende Indexvariable *MD_xxx* einen Wert größer –1 hat. Wenn ja, kann davon ausgegangen werden, das ein solches Gerät angeschlossen ist und funktioniert und TRUE wird zurückgegeben.
TPsdParameters::Dlg_OnInit(), *c_layer.cpp::dllsDeviceValid()*

IsValidId

Überprüft, ob für den angegebenen Parameter ein Gerät im Geräte-Array existiert.
m_main.cpp::FrameWndProc()

LastId

Gibt *nLastDevice* zurück.
TDList::InitializeModule(), *TScsParameters::TScsParameters()*, *TCommonDevParam::Dlg_OnInit()*, ***TSetupAreaScan::Dlg_OnInit()***, ***TSetupStepScan::Dlg_OnInit()***, ***TSetupContinuousScan::Dlg_OnInit()***, ***TTopographySetParam::Dlg_OnInit()***

ParsingAxis

Versucht, in dem als Parameter übergebenen String die Bezeichnung für eine Geräteklasse zu erkennen und gibt bei Erfolg die entsprechende Geräteklasse zurück, andernfalls die Geräteklasse für ein *GenericDevice*.
TDList::InitializeModule(), *c_layer.cpp::dlParsingDevice()*

SaveModuleSettings

Gibt TRUE zurück. (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

SetDevice

Legt das Gerät mit dem Index, der als Parameter übergeben wurde, als das aktive fest.

TCommonDevParam::Dlg_OnCommand(), ***TAreaScan::ShowSensorContinuous()***, ***TAreaScan::CalibratePsd()***, ***TAreaScan::ExternSychronized()***, ***TCounterWindow::TCounterWindow()***, ***TCounterWindow::rButtonDown()***, ***TCounterWindow::SetFocus()***, ***TScan::InitializeTask()***, *c_layer.cpp::dlSetDevice()*, *m_main.cpp::DoCommandsFrame()*

SetParametersDlg

Erzeugt einen neuen *TCommonDevParam*-Dialog für das aktive Gerät und führt ihn aus. ***m_main.cpp::DoCommandsFrame()***, ***m_main.cpp::DoCommandsChild***

8. Fehler, Probleme

Es gibt keinerlei Vorkehrung, die dafür sorgt, daß es im System genau ein *TDList*-Objekt gibt.

CalibrateDlg

CalibrateDlg wird nur einmal in einem auskommentierten Bereich verwandt.

DeleteDevices

DeleteDevices wird lediglich einmal in *m_main.cpp* aufgerufen, um die einzelnen Geräte ordentlich abzuschließen und sie dann löschen zu lassen. Diese Absicht wurde offensichtlich nicht weiter ausgeführt.

GetModuleHandle

GetModuleHandle wird nie aufgerufen.

InitializeModule

InitializeModule liest aus dem ini-File die Gerätekonfiguration. Die dazu verwendete Routine arbeitet ineffizient, da sie wiederholt die Darstellung einer Zeichenkette in Großbuchstaben für einen Vergleich erzeugt, um sie beim nächsten Vergleich neu zu ermitteln.
Sollte InitializeModule einen Detektor nicht initialisieren können, so wird dieser aus der Liste entfernt und durch ein TDevice-Objekt ersetzt. Dieser trägt als Bezeichnung (Characteristic) weiterhin diejenige, die diesem Detektor im ini-File ursprünglich zugedacht war. Dieser Name wird auch weiterhin in Auswahlboxen angezeigt, auch wenn das Gerät nicht funktioniert und es inhärent sinnlos ist, dann dafür Einstellungen vornehmen zu wollen.

SaveModuleSettings

SaveModuleSettings wird nie aufgerufen.

MD_xx

Der MD_xx Indizierungsmechanismus wird zwar von den Counters beschrieben, allerdings nie gelesen bzw. extern verwandt.

class TDevice

1. UML-Klassendiagramm

class TDevice	
	+ bAnalysator : BOOL + bDataValid : BOOL # bDebug : BOOL # bDeviceActive : BOOL # bDeviceOk : BOOL + bDeviceOpen : BOOL + bDiskriminator : BOOL + bSound : BOOL + bStaticFailure : BOOL + Characteristic[] : char + dRealTime : double # dwElapsedTime : DWORD # dwExposureCounts : DWORD # dwLowerCountBound : DWORD # dwStartTime : DWORD # dwUpperCountBound : DWORD # eHardware : THardware # fExposureTime : float # fFailure : float + fIntensity : float # fLowerTimeBound : float + fSigma : float # fUpperTimeBound : float # hControlWnd : HWND # hDisplayWnd : HWND # hWndFrame : HWND - nDevId + nFunctionId : int <<static>> - nMaxId : int + nPeakLevel : int + nPeakWindow : int # Settings : TDSettings # uAskTime : UINT # wpJumpTo : WPARAM
<<virtual>>	+ ~TDevice(void)
	# CalcTime(DWORD) : double
<<static>> <<CALLBACK>>	+ EventHandler(UINT, UINT, DWORD, DWORD, DWORD)
<<virtual>>	+ GetAddedChannels(void) : int
<<virtual>>	+ GetAngleRange(TUnitType) : float
<<virtual>>	+ GetAngleRange(void) : float
<<virtual>>	+ GetAngleStep(void) : float
<<virtual>>	+ GetData(float&) : int
<<virtual>>	+ GetData(TCurve&) : int
<<virtual>>	+ GetData(WORD*, WORD, WORD) : int
	# GetDisplay(void) : HWND
	+ GetExposureValues(float&, DWORD&, float&)
	+ GetHardware(void) : THardware

<<inline>>	+	GetId(void) : int
<<virtual>>	+	GetMaximumChannel(void) : UINT
<<virtual>>	+	GetMaximumIntensity(void) : float
	+	GetMFunction(void) : int
	+	GetSigma(void) : float
<<virtual>>	+	GetUnitStr(void) : LPCSTR
<<virtual>>	+	GetWidth(TUnitType) : float
<<virtual>>	+	GetWidth(void) : float
<<virtual>>	+	Initialize(void) : int
<<virtual>>	+	InitializeEvent(HWND, int) : BOOL
	+	IsActive(void) : BOOL
<<virtual>>	+	IsHardOverflow(void) : BOOL
<<inline>>	+	IsOk(void) : int
<<virtual>>	+	IsSoftOverflow(void) : BOOL
<<virtual>>	+	MeasureStart(void) : int
<<virtual>>	+	MeasureStop(void) : int
<<virtual>>	+	MeasureStopExternal(void) : int
<<virtual>>	+	PollDevice(void) : int
<<virtual>>	+	PopSettings(void) : BOOL
<<virtual>>	+	PushSettings(void) : BOOL
	+	SelectMFunction(int) : BOOL
<<virtual>>	+	SetAddedChannels(int)
<<virtual>>	+	SetAngleStep(float)
<<virtual>>	+	SetDataType(TPsdDataType)
	#	SetDisplay(HWND)
	+	SetExposureValues(float, DWORD, float) : int
	+	SetHost(HWND)
<<virtual>>	+	SetParameters(void) : int
<<virtual>>	+	SetSignalGrowUp(BOOL)
<<virtual>>	+	SetSound(void) : BOOL
<<virtual>>	+	SetSpezificParametersDlg(void)
<<virtual>>	+	SingleStepMeasurement(int) : int
	+	Tdevice(void)
	+	UpdateDisplay(void)

2. Klassenhierarchie

TDevice ist Wurzelklasse für alle Detektorklassen.

3. Friends

class TDList;
class TCommonDevParam;
class TCounterWindow;
class TAreaScan;

4. Files

Klassendefinition: m_devcom.h
Implementation: counters.cpp

5. Verantwortlichkeiten der Klasse

TDevice ist Wurzelklasse für alle Detektorklassen und gleichzeitig auch das Interface zu allen Detektoren. Sie definiert grundlegende Methoden bzw. implementiert diese teilweise (Testdetektor).

6. Beschreibung der Attribute

bAnalysator

Die Funktion von bAnalysator ist nicht erkennbar. (Ist dem in keinem Dialog definierten Button *id_Analysator* zugeordnet.)
TDevice::TDevice(), ***TCommonDevParam::Dlg_OnCommand()***

bDataValid

Status-Flag, das dann gesetzt ist, wenn gültige Meßdaten vorliegen.
TDevice::TDevice(), *TDevice::InitializeEvent()*, *TDevice::GetData()*, *TDevice::MeasureStart()*, *TDevice::PollDevice()*, *TGenericDevice::MeasureStop()*, *TGenericDevice::PollDevice()*, *TEncoder::GetData()*, *TEncoder::PollDevice()*, *TPsd::MeasureStart()*, *TPsd::MeasureStop()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::MeasureStop()*, *TRadicon::PollDevice()*, *TRadicon::InitializeEvent()*, ***TAreaScan::CounterSetRequest()***

bDebug

Status-Flag, das dann gesetzt ist, wenn der Debugging-Modus aktiviert ist.
TDevice::Initialize(), *TDevice::MeasureStart()*, *TDevice::MeasureStop()*, *TDevice::PollDevice()*, *TPsd::GetData()*, *TStoe_Psd::PsdStop()*, *TRadicon::MeasureStart()*, *TRadicon::MeasureStop()*, *TRadicon::PollDevice()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::BuildOperation()*, *TBraun_Psd::ResetDelayTime()*, *TBraun_Psd::Look_till_BaseAddr1()*, *TBraun_Psd::SynchronHexFile()*, *TBraun_Psd::LoadHexFile()*

bDeviceActive

Status-Flag, das dann gesetzt ist, wenn eine Messung läuft.
TDevice::TDevice(), *TDevice::MeasureStart()*, *TDevice::MeasureStop()*, *TDevice::PollDevice()*, *TDevice::PopSettings()*, *TDevice::IsActive()*, *TGenericDevice::MeasureStart()*, *TGenericDevice::MeasureStop()*, *TGenericDevice::PollDevice()*, *TEncoder::PollDevice()*, *TPsd::MeasureStart()*, *TPsd::MeasureStop()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::Inizialize()*, *TRadicon::InizializeEvent()*, *TRadicon::MeasureStart()*, *TRadicon::MeasureStop()*, *TRadicon::PollDevice()*,

bDeviceOk

Status-Flag, das dann gesetzt ist, wenn kein Fehler aufgetreten ist.
TDevice::TDevice(), *TDevice::IsOk()*, *TEncoder::Initialize()*, *TPsd::Initialize()*, *TRadicon::Initialize()*, *TStoe_Psd::PollDevice()*, *TPsd::PollDevice()*, *TBraun_Psd::TBraun_Psd()*, *TBraun_Psd::PsdInit()*

bDeviceOpen

Status-Flag, das dann gesetzt ist, wenn „das Gerät in irgend einer Form aktiv ist“.
TDevice::TDevice(), ***TCommonDevParam::Dlg_OnCommand()***, ***TMacroExecute::Dlg_OnCommand()***, ***TAdjustmentExecute::LeaveDialog()***, ***TCounterWindow::~TCounterWindow()***, ***TCounterWindow::Create()***, ***TTopographyExecute::Dlg_OnInit()***

bDiskriminator

Die Funktion von bDiskriminator ist nicht erkennbar. (Wird in *TDevice::TDevice()* auf den negierten Wert von *bAnalysator* gesetzt und sonst nirgendwo geschrieben bzw. gelesen – scheint also keine relevante Funktion zu besitzen.)

bSound

Status-Flag, das dann gesetzt ist, wenn ein akustisches Feedback gewünscht wird.
TDevice::TDevice(), *TDevice::~TDevice()*, *TGenericDevice::SetSound()*, *TRadicon::TRadicon()*, *TRadicon::~TRadicon()*, *TRadicon::SetParameters()*, ***TCommonDevParam::Dlg_OnCommand()*, *TCommonDevParam::CanClose()***

bStaticFailure

Status-Flag, das dann gesetzt ist, wenn der Meßfehler auf einem bestimmten Wert gehalten werden soll. *bStaticFailure* kann bisher nur in einem Dialogfenster eingestellt werden und wird sonst nicht verwandt.
TDevice::TDevice(), *TDevice::PushSettings()*, *TDevice::PopSettings()*, ***TCommonDevParam::Dlg_OnCommand()*, *TCommonDevParam::CanClose()***

Characteristic

Beinhaltet einen Kennstring für das Gerät.
TDevice::TDevice, TDList::InitializeModule(), ***TAreaScan::InitializeTask()***, ***TSetupAreaScan::Dlg_OnInit()*, *TCounterWindow::SetTitle()*, *TSetupStepScan::Dlg_OnInit()***, ***TSetupStepScan::Dlg_OnCommand()***, ***TCommonDevParam::Dlg_OnInit()***, ***TCommonDevParam::Dlg_OnCommand()***, ***TSetupContinuousScan::Dlg_OnInit()*, *TSetupContinuousScan::Dlg_OnCommand()***, ***TTopographySetParam::Dlg_OnInit()*, *TTopographySetParam::Dlg_OnCommand()***

dRealTime

Beinhaltet die tatsächliche Meßzeit.
TPsd::GetData(), *TPsd::PollDevice()*, *TPsd::PollDevice()*, *TPsd::GetMaximumIntensity()*, *TStoe_Psd::PollDevice()*, ***TAreaScan::CounterSetRequest()***

dwElapsedTime

Beinhaltet die seit Meßbeginn verstrichene Zeit.
TPsd::MeasureStopExternal(),*TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*

dwExposureCounts

Beinhaltet die obere Grenze für die Impulsanzahl.
TDevice::TDevice(), *TDevice::~TDevice()*, *TDevice::SetExposureValues()*,*TDevice::GetExposureValues()*, *TDevice::Initialize()*, *TDevice::PushSettings()*, *TDevice::PopSettings()*, *TGenericDevice::Initialize()*, *TGenericDevice::SetParameters()*, *TGenericDevice::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*, *TRadicon::SetParameters()*, ***TCommonDevParam::Dlg_OnCommand()***, ***TCommonDevParam::CanClose()***,

dwLowerCountBound

Beinhaltet den kleinsten möglichen Wert, den dwExposureCounts annehmen kann.
TDevice::TDevice(), ***TCommonDevParam::CanClose()***

dwStartTime

Beinhaltet den Zeitpunkt, an dem die Messung gestartet wurde. Wird für die Berechnung der bereits verstrichenen Meßzeit benötigt.
TPsd::MeasureStart(), *TPsd::MeasureStopExternal()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*,*TRadicon::PollDevice()*

dwUpperCountBound

Beinhaltet den größten möglichen Wert, den dwExposureCounts annehmen kann.
TDevice::TDevice(), ***TCommonDevParam::CanClose()***

eHardware

Beinhaltet den Typ des Gerätes.
TDevice::Initialize(), *TGenericDevice::Initialize()*, *TEncoder::Initialize()*, *TPsd::Initialize()*, *TStoe_Psd::TStoe_Psd()*, *TStoe_Psd::Initialize()*, *TRadicon::Initialize()*, *TBraun_Psd::Initialize()*, *TDevice::GetHardware()*

fExposureTime

Beinhaltet die obere Grenze für die Meßzeit.
TDevice::TDevice(), *TDevice::~TDevice()*, *TDevice::SetExposureValues()*, *TDevice::GetExposureValues()*, *TDevice::Initialize()*, *TDevice::InitializeEvent()*, *TDevice::MeasureStart()*, *TDevice::PushSettings()*, *TDevice::PopSettings()*, *TGenericDevice::Initialize()*, *TGenericDevice::SetParameters()*, *TGenericDevice::PollDevice()*, *TEncoder::Initialize()*, *TPsd::MeasureStart()*, *TPsd::MeasureStopExternal()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*, *TRadicon::SetParameters()*, *TRadicon::InitializeEvent()*, ***TCommonDevParam::Dlg_OnCommand()***, ***TCommonDevParam::CanClose()***

fFailure

Beinhaltet den anzunehmenden Meßfehler.
TDevice::TDevice(), TDevice::SetExposureValues(), *TDevice::GetExposureValues()*, *TDevice::PushSettings()*, *TDevice::PopSettings()*, *TGenericDevice::Initialize()*, ***TCommonDevParam::Dlg_OnCommand()***, ***TCommonDevParam::CanClose()***

fIntensity

Beinhaltet die aus den Meßwerten berechnete Intensität.
TDevice::TDevice(), *TDevice::GetData()*, *TDevice::PollDevice()*, *TGenericDevice::PollDevice()*, *TEncoder::GetData()*, *TEncoder::PollDevice()*, *TPsd::GetData()*, *TPsd::GetData()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*, *c_layer.cpp::InquireIntensity_SCS()*

fLowerTimeBound

Beinhaltet den kleinsten möglichen Wert, den fExposureTime annehmen kann.
TDevice::TDevice(), *TGenericDevice::Initialize()*, *TRadicon::TRadicon()*, ***TCommonDevParam::CanClose()***

fSigma

Ist ein Wert, der rechnerisch aus der Intensität gewonnen wird. (Wird von ***TGotoPeakCmd::ControlStep()*** verwendet.)
TDevice::TDevice(), *TDevice::GetData()*, *TDevice::PollDevice()*, *TPsd::GetData()*, *TPsd::GetData GetSigma()*

fUpperTimeBound

Beinhaltet den größten möglichen Wert, den fExposureTime annehmen kann.
TDevice::TDevice(), *TGenericDevice::Initialize()*, *TRadicon::TRadicon()*, ***TCommonDevParam::CanClose()***

hControlWnd

Handle für das Fenster, welches die Daten vom Meßvorgang bezieht.
TDevice::TDevice(), *TDevice::InitializeEvent()*, *TDevice::MeasureStart()*, *TDevice::PollDevice()*, *TGenericDevice::MeasureStart()*, *TGenericDevice::PollDevice()*, *TEncoder::PollDevice()*, *TPsd::MeasureStart()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*, *TRadicon::InitializeEvent()*, *TDevice::SetHost()*, ***TCounterWindow::TCounterWindow()***, ***TCounterWindow::CounterSetRequest()***

hDisplayWnd

Handle für das Fenster, welches die Meßdaten anzeigt.
TDevice::TDevice(), *TDevice::UpdateDisplay()*, *TDevice::MeasureStart()*, *TDevice::PollDevice()*, *TGenericDevice::MeasureStart()*, *TGenericDevice::PollDevice()*, *TEncoder::PollDevice()*, *TPsd::MeasureStart()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*, *TDevice::SetDisplay()*, *TDevice::GetDisplay()*, ***TCounterWindow::~TCounterWindow()***, ***TCounterWindow::CanOpen()***, ***TCounterWindow::Create()***

hWndFrame

Handle für das Fenster, innerhalb dessen die Meßvorgänge angezeigt werden (Hauptfenster).
TDList::InitializeModule(), *TDevice::TDevice()*, *TDevice::MeasureStart()*, *TDevice::MeasureStop()*, *TDevice::PollDevice()*, *TGenericDevice::MeasureStart()*, *TGenericDevice::MeasureStop()*, *TGenericDevice::PollDevice()*, *TEncoder::PollDevice()*, *TPsd::MeasureStart()*, *TPsd::MeasureStop()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*

nDevId

ID der TDevice – Instanz (ID des Geräts).
TDevice::TDevice(), *TDevice::~TDevice()*, *TDevice::Initialize()*, *TDevice::GetId()*

nFunctionId

Beinhaltet den aktuellen Betriebsmodus bzw. -zustand des Gerätes.
TDevice::TDevice(), *TDevice::PushSettings()*, *TDevice::PopSettings()*,*TDevice::SelectMFunction()*, *TDevice::GetMFunction()*

nMaxId

Anzahl von TDevice-Objekten die bisher instantiiert wurden.
TDevice::TDevice(), *TDevice::~TDevice()*

nPeakLevel

Die Funktion von *nPeakLevel* ist nicht erkennbar. (Wird in *TDevice::TDevice()* mit einem Wert initialisiert und danach nie mehr gelesen bzw. geschrieben – scheint also keine relevante Funktion zu besitzen.)
TDevice::TDevice()

nPeakWindow

Die Funktion von *nPeakWindow* ist nicht erkennbar. (Wird in *TDevice::TDevice()* mit einem Wert initialisert und danach nie mehr gelesen bzw. geschrieben – scheint also keine relevante Funktion zu besitzen.)
TDevice::TDevice()

Settings

TDeviceSettings-Struktur, welche die Einstellungen des Gerätes speichern soll. Sie kann mit *PushSettings()* mit allen relevanten Variablen beschrieben und mit *PopSettings()* in alle entsprechenden Variablen ausgelesen werden.
TDevice::PushSettings(), *TDevice::PopSettings()*, *TPsd::PushSettings()*, *TPsd::PopSettings()*

uAskTime

Zeit, die ein Meßtimer zu laufen hat.
TDevice::TDevice(), *TDevice::MeasureStart()*, *TGenericDevice::TGenericDevice()*, *TGenericDevice::MeasureStart()*, *TGenericDevice::PollDevice()*, *TPsd::MeasureStart()* *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*

wpJumpTo
wpJumpTo wird einmal auf die Nachricht *cm_CounterSet* gesetzt und im weiteren Verlauf als Parameter für *PostMessage* verwandt.
TDevice::TDevice(), *TDevice::PollDevice()*, *TGenericDevice::PollDevice()*, *TEncoder::PollDevice()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::PollDevice()*

7. Beschreibung der Methoden

TDevice
Initialisiert Attribute mit Standardwerten. Der Klassenzähler für die TDevice-Objekte wird inkrementiert.
Allgemein: Konstruktor
TDList::InitializeModule()

~TDevice
Schreibt die aktuellen Einstellungen für das Gerät zurück in die ini-Datei und dekrementiert den TDevice-Klassenzähler.
Allgemein: Destruktor
TDList::~TDList(), *TDList::InitializeModule()*

CalcTime
Allgemein: Rechnet Millisekunden in Sekunden um.
TPsd::MeasureStopExternal(), *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TRadicon::MeasureStart()*, *TRadicon::PollDevice()*

EventHandler
Berechnet die Werte für den Test-Zähler.
Die Klassenvariable *nCalledEvents* wird bei jedem Aufruf inkrementiert, bis *nCalledEvents* gleich *nEventCalls* ist. Dann wird der Event-Timer zerstört. Allgemein: Callback-Methode, welche bei Ablauf eines bestimmten Timers aufgerufen werden soll.
TDevice::InitializeEvent()

GetAddedChannels
Gibt 1 zurück.
Allgemein: Liefert die Anzahl der Kanäle, deren Meßwerte zu einem gemeinsamen Meßwert zusammengefaßt werden (*TPsd::nAddedChannels*). (1-dimensionale Detektoren)
TAreaScanParameters::TAreaScanParameters(), TSetupAreaScan::Dlg_OnCommand()

GetAngleRange(void)
Gibt 1 zurück.
Allgemein: *GetAngleRange(void)* ruft lediglich *GetAngleRange(TUnitType eunit)* mit dem richtigen Parameter auf.
TAreaScan::SetRanges(), ***TPsdParameters::Dlg_OnCommand()***

GetAngleRange(TUnitType)
Gibt 1 zurück.
Allgemein: *GetAngleRange(TUnitType eunit)* berechnet den gemessenen Winkelbereich. Abhängig vom Parameter *eunit* (Grad, Minuten) wird *TPsd::fAngleStep* in Bogensekunden umgerechnet und mit der Anzahl der Kanäle multipliziert. (1-dimensionale Detektoren)
TAreaScanParameters::TAreaScanParameters(), ***TAreaScan::InitializeTask()***, ***TAreaScan::SetRanges()***,

GetAngleStep
Gibt 0.0 zurück.

Allgemein: Liefert den Winkel, der von einem Kanal abgedeckt wird (*TPsd::fAngleStep*).
(1-dim. Detektoren)
TAreaScan::SetRanges(), ***TCalibratePsd::Dlg_OnCommand()***,
TCalibratePsd::CanClose(), ***TPsd::PsdReadOut()***

GetData(float&)

Allgemein: Gibt den Wert von *fIntensity*, also den gemessenen Wert zurück.
TAreaScan::CounterSetRequest(), ***TSteering::DeviceRequest()***,
TAdjustmentWindow::CounterSetRequest(), ***TCounterWindow::CounterSetRequest()***,
TTopographyExecute::Dlg_OnCommand(), ***TScan::CounterSetRequest()***

GetData(TCurve&)

Gibt 0 zurück.
Allgemein: Gibt den Meßwert in einem *TCurve* – Objekt zurück.
TAreaScan::CounterSetRequest(),

GetData(WORD*, WORD, WORD)

Gibt 0 zurück. (Wird nie aufgerufen.)

GetDisplay

Allgemein: Gibt *hDisplayWnd* zurück.
TAreaScan::CalibratePsd(), ***TAreaScan::ExternSychronized()***

GetExposureValues

Allgemein: Gibt die aktuellen Einstellungen für die Meß-Limits zurück.
TPsdRemoteSync::TPsdRemoteSync(), ***TSteering::ToggleInterrupt()***,
c_layer.pp::dGetExposureValues()

GetHardware

Allgemein: Gibt den Wert von *eHardware* zurück.
TDList::~TDList(), *TScsParameters::TScsParameters()*

GetId

Allgemein: Gibt die Instanz ID von *TDevice* zurück (*nDevId*).
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::SetEnergyRange()*,
TDevice::MeasureStart(), *TDevice::MeasureStop()*, *TDevice::PollDevice()*,
TGenericDevice::TGenericDevice() TGenericDevice::MeasureStart(),
TGenericDevice::MeasureStop(), *TGenericDevice::PollDevice()*, *TEncoder::PollDevice()*,
TPsd::~TPsd(), *TPsd::Initialize()*, *TPsd::MeasureStart()*, *TPsd::MeasureStop()*,
TPsd::PollDevice(), *TStoe_Psd::PollDevice()*, *TRadicon::TRadicon()*,
TRadicon::~TRadicon(), *TRadicon::MeasureStart()*, *TRadicon::MeasureStop()*,
TRadicon::PollDevice(),
TAreaScanParameters::SetDevice(), ***TAreaScan::InitializeTask()***,
TCalibratePsd::CanClose(), ***TSetupAreaScan::Dlg_OnCommand()***,
TMacroExecute::Dlg_OnCommand(), ***TAdjustmentExecute::Dlg_OnInit()***,
TCounterWindow::CounterSetRequest(),
TCounterWindow::SetupLogging(),***TCounterWindow::rButtonDown()***,
TCounterWindow::SetFocus(), ***TTopographyExecute::Dlg_OnInit()***,
TTopographyExecute::Dlg_OnTimer(), ***TTopographyExecute::Dlg_OnCommand()***,
TScan::InitializeTask(), ***TSetupStepScan::Dlg_OnCommand()***,
TAngleControl::Dlg_OnCommand(), ***TAngleControl::LeaveDialog()***,
TCommonDevParam::Dlg_OnCommand()

GetMaximumChannel

Gibt 0 zurück.
Allgemein: Liefert die Nummer des Kanals, auf dem der größte Meßwert registriert wurde.
(1-dim. Detektoren)
TAreaScan::CounterSetRequest()

GetMaximumIntensity

Gibt 1.0 zurück.
Allgemein: Liefert den Durchschnitt der größten auf einem der Kanäle gemessenen Einzelintensität über der Meßzeit (*dwMaxCounts / dRealTime*).
TAreaScan::CounterSetRequest()

GetMFunction

Allgemein: Gibt den Wert von *nFunctionId* zurück. (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

GetSigma

Allgemein: Gibt den Wert von *fSigma* zurück. (Wird in *TGotoPeakCmd::ControlStep()* verwendet. *m_steerg.h* kapselt diese Funktion über eine dort definierte Funktion *GetSigma()*.

GetUnitStr

Gibt den leeren String “” zurück.
Allgemein: Liefert die Maßeinheit der Winkelangaben im Klartext (*TPsd::szUnit*).
(1-dimensionale Detektoren.)

GetWidth(void)

Gibt 1 zurück.
Allgemein: Liefert den Winkel der von einer Kanalgruppe abgedeckt wird
(*TPsd::fAngleWidth*). (1-dimensionale Detektoren)
TAreaScan::GetShift(), ***TAreaScan::SaveMeasurementInfo()***,
TSetupAreaScan::Dlg_OnCommand()

GetWidth(TUnitType)

Gibt 1 zurück.
Allgemein: Liefert den Winkel der von einer Kanalgruppe abgedeckt wird entsprechend dem übergebenen Parameter in Grad bzw. Minuten. (1-dimensionale Detektoren)
TSetupAreaScan::Dlg_OnCommand()

Initialize

Initialisiert die Meß-Limits mit den Werten aus dem ini-File.
Allgemein: Initialisierung von Detektorattributen mit Werten aus dem ini-File.
TDList::InitializeModule(), *TGenericDevice::Initialize()*, *TEncoder::Initialize()*, *TPsd::Initialize()*, *TRadicon::Initialize()*

InitializeEvent

Setzt die Klassenvariable *nCalledEvents* auf 0 und setzt *nEventCalls* auf einen Endwert. Dann wird ein Event-Timer gestartet, der alle *fExposureTime*-Sekunden *EventHandler()* ausführt.
Allgemein: Initialisierung von *EventHandler()*.
TScan::InitializeTask()

IsActive(void)

Allgemein: Gibt den Wert von *bDeviceActive* zurück.
TDevice::SetExposureValues(), ***TTopographyExecute::Dlg_OnCommand()***,
TCommonDevParam::Dlg_OnCommand(), ***TCommonDevParam::LeaveDialog()***

IsHardOverflow

Gibt FALSE zurück.
Allgemein: Liefert das Attribut *TPsd::bHardOverflow*, welches anzeigt, daß im Gerät ein Überlauf stattgefunden hat. (1-dimensionale Detektoren) (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen. *bHardOverflow* wird nur innerhalb der TDevice-Klassen ausgewertet.)

IsOk

Allgemein: Gibt den Wert von *bDeviceOk* zurück. (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen. *bDeviceOk* wird nur innerhalb der TDevice-Klassen ausgewertet.)

IsSoftOverflow

Gibt FALSE zurück.
Allgemein: Liefert das Attribut *TPsd::bSoftOverflow*, welches anzeigt, daß während einer Messung ein Meßwert aus gemessener Intensität und verstrichener Zeit berechnet worden ist, der größer als *TPsd::OverflowIntensity* ist.
TAreaScan::CounterSetRequest()

MeasureStart

Startet den Messvorgang. Dazu wird ein Timer gestartet, der nach *fExposureTime* abläuft. Das Gerät wird als aktiv markiert und es wird festgelegt, daß im Moment keine gültigen Daten vorliegen.
TDevice::SetExposureValues(), *TDevice::PopSettings()*,
TCommonDevParam::Dlg_OnCommand(), TCommonDevParam::LeaveDialog(),
TAreaScan::~TAreaScan(), TAreaScan::CounterSetRequest(),
TAreaScan::ShowSensorContinuous(), ***TAreaScan::CalibratePsd()***,
TAreaScan::ExternSychronized(), ***TCalibratePsd::Dlg_OnInit()***,
TCalibratePsd::Dlg_OnCommand(), ***TSteering::ToggleInterrupt()***, ***TSteering::WakeUp()***,
TSteering::DeviceRequest(), ***TSteering::NotifyCmdReady()***,
TSteering::NotifyMacroReady(),***TMacroExecute::Dlg_OnCommand()***,
TAdjustmentExecute::LeaveDialog(), ***TCounterWindow::Create()***,
TCounterWindow::CounterSetRequest(), ***TCounterShowParam::CanClose()***,
TTopographyExecute::Dlg_OnCommand(), ***TTopographyExecute::LeaveDialog()***,
TScan::~TScan(), ***TScan::SteeringReady()***, ***TAngleControl::LeaveDialog()***,
c_layer.cpp::dMeasureStart(), ***m_main.cpp::FrameWndProc()***

MeasureStop

Prüft zuerst, ob überhaupt eine Messung läuft. Wenn nicht, ist die Arbeit beendet.
Der Meß-Timer wird gestoppt und das Gerät als inaktiv markiert.
TDevice::SetExposureValues(), *TDevice::MeasureStart()*, *TDevice::PopSettings()*, ***TCommonDevParam::Dlg_OnCommand(), TCommonDevParam::LeaveDialog()***, ***TAreaScan::~TAreaScan(), TAreaScan::CounterSetRequest(), TAreaScan::ShowSensorContinuous()***, ***TAreaScan::InitializeTask()***, ***TPsdRemoteSync::Dlg_OnInit()***, ***TPsdRemoteSync::LeaveDialog()***, ***TCalibratePsd::Dlg_OnInit()***, ***TCalibratePsd::Dlg_OnCommand()***, ***TCalibratePsd::LeaveDialog()***, ***TSteering::StartCmdExecution()***, ***TSteering::ToggleInterrupt()***, ***TSteering::StartMacroExecution()***, ***TSteering::NotifyCmdReady()***, ***TSteering::NotifyMacroReady()***, ***TMacroExecute::Dlg_OnCommand()***, ***TCounterWindow::Create()***, ***TCounterWindow::TCounterWindow()***, ***TCounterShowParam::CanClose()***, ***TTopographyExecute::Dlg_OnCommand()***, ***TTopographyExecute::LeaveDialog()***, ***TScan::~TScan()***, ***TScan::InitializeTask()***, ***TScan::SteeringReady()***, ***TAngleControl::LeaveDialog()***, ***c_layer.cpp::dMeasureStop()***, ***m_main.cpp::FrameWndProc()***

MeasureStopExternal

Gibt 1 zurück.
Allgemein: Berechnet die seit Meßbeginn verstrichene Zeit (*dwElapsedTime*) und die obere Grenze für die Meßzeit (*fExposureTime*). (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

PollDevice

Stellt zuerst fest, ob noch eine Messung läuft. Wenn ja, können keine Daten ermittelt werden und PollDevice endet mit Fehler. Nun wird der Meß-Timer gestoppt, der eigentlich nicht mehr existieren sollte, wenn die Messung mit *MeasureStop()* beendet wurde. Der Wert wird aus *Omega*, *Psi* und *fIntensit*y berechnet (wesentliche Funktion des Testdetektors).
Abschließend wird *bDeviceActive* auf FALSE gesetzt, was unlogisch ist, da der Programmablauf diese Stelle nie erreicht hätte, wenn *bDeviceActive* einen anderen Wert als FALSE gehabt hätte. !!!
Allgemein: Berechnet den Wert von *fIntensity* auf Grundlage der aktuellen Detektormeßwerte.
m_main.cpp:: FrameWndProc()

PopSettings

Schreibt die entsprechenden Werte aus der Struktur *TDSettings Settings* in die Attribute *fExposureTime*, *fFailure*, *dwExposureCounts*, *bStaticFailure* und *nFunctionId*. Ist die Messung aktiv, so wird sie gestoppt, *SetParameters()* aufgerufen und anschließend die Messung wieder gestartet. Andernfalls wird nur *SetParameters()* aufgerufen.
Allgemein: Übertragen der Werte der Detektorattribute *fExposureTime*, *fFailure*, *dwExposureCounts*, *bStaticFailure* und *nFunctionId* aus der *TDSettings*-Struktur in die entsprechenden Objektvariablen.
TPsd::PopSettings(), ***TAreaScan::SteeringReady()***, ***TAreaScan::ShowSensorContinuous()***, ***TSteering::ToggleInterrupt()***, ***TSteering::NotifyCmdReady()***, ***TSteering::NotifyMacroReady()***, ***TTopographyExecute::LeaveDialog()***, ***TScan::SteeringReady()***, ***TAngleControl::LeaveDialog()***

PushSettings

Allgemein: Schreibt die Werte der Attribute *fExposureTime*, *fFailure*, *dwExposureCounts*, *bStaticFailure* und *nFunctionId* in eine *TDSettings*-Struktur.
TDevice::Initialize(), *TPsd::PushSettings()*, *TPsd::Initialize()*,
TCommonDevParam::CanClose()

SelectMFunction

Allgemein: Setzt den aktuellen Betriebszustand (*nFunctionId*) auf den Wert des übergebenen Parameters und gibt TRUE zurück (Es wird nur der Wert des Attributes verändert und keine wirkliche Konfiguration des Detektors vorgenommen.) (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

SetAddedChannels

Allgemein: Berechnet den von einer Kanalgruppe abgedeckten Winkel (*TPsd::fAngleWidth*) entsprechend dem übergebenen Parameter. (1-dimensionale Detektoren)
TPsd::Initialize(),*TPsdParameters::CanClose()*, ***TAreaScan::InitializeTask()***,
TCalibratePsd::Dlg_OnInit(), ***TCalibratePsd::Dlg_OnCommand()***,
TSetupAreaScan::TSetupAreaScan(), ***TSetupAreaScan::Dlg_OnCommand()***

SetAngleStep

Allgemein: Setzt den Winkel der von einem Kanal abgedeckt wird (*TPsd::fAngleStep*) auf den Wert des übergebenen Parameters. (Es wird nur der Attributwert verändert.) (1-dimensionale Detektoren)
TCalibratePsd::Dlg_OnCommand()

SetDataType

Allgemein: Setzt das Attribut *TBraun_Psd::eDataType*. (nur BraunPsd)
TBraun_Psd::TBraun_Psd(), ***TAreaScan::ShowSensorContinuous()***

SetDisplay

Allgemein: Setzt das Attribut *hDisplayWnd* auf den übergebenen Parameter.
TAreaScan::~TAreaScan(), ***TAreaScan::CalibratePsd()***,
TAreaScan::ExternSychronized()

SetExposureValues

Allgemein: Setzt die Meß-Limits für den Zählvorgang. *fExposureTime*, *dwExposureCounts* und *fFailure* werden gesetzt. Falls der Messvorgang aktiv ist, so wird die Messung beendet, die Limits gesetzt und die Messung neu gestartet.
TGenericDevice::Initialize(), ***TAreaScan::InitializeTask()***,
TPsdRemoteSync::Dlg_OnCommand(), ***TPsdRemoteSync::LeaveDialog()***,
TCalibratePsd::Dlg_OnInit() TSteering::ToggleInterrupt(),
TAdjustmentExecute::GetNextTask(), ***TTopographyExecute::Dlg_OnCommand()***,
TScan::InitializeTask(), ***c_layer.cpp:: dSetExposureValues()***

SetHost

Allgemein: Setzt das Attribut *hControlWnd* auf den übergebenen Parameter.
TAreaScan::~TAreaScan(), TAreaScan::InitializeTask(), TAreaScan::SteeringReady(), TAreaScan::ShowSensorContinuous(), TPsdRemoteSync::Dlg_OnInit(), TPsdRemoteSync::LeaveDialog(), TCalibratePsd::Dlg_OnInit(), TCalibratePsd::LeaveDialog(), TSteering::StartCmdExecution(), TSteering::ToggleInterrupt(), TSteering::StartMacroExecution(), TSteering::NotifyCmdReady(), TSteering::NotifyMacroReady(), TTopographyExecute::LeaveDialog(), TScan::~TScan(), TScan::SteeringReady()

SetParameters
Gibt 1 zurück.
Allgemein: Setzen der Meß-Limits.
TDevice::SetExposureValues(), TDevice::PopSettings(), TRadicon::Initialize(), TRadicon::SetSound(), TCommonDevParam::Dlg_OnCommand(), TCommonDevParam::LeaveDialog(), TAreaScan::InitializeTask(), TCounterWindow::Create()

SetSignalGrowUp
Allgemein: Setzt das Attribut *TPsd::bSignalGrowUp* entsprechend dem übergebenen Parameter.
(1-dimensionale Detektoren)
TAreaScan::InitializeTask(), TAreaScan::ShowSensorContinuous(), TAreaScan::CalibratePsd(), TAreaScan::ExternSychronized(), TPsdRemoteSync::Dlg_OnInit()

SetSound
Gibt TRUE zurück.
Allgemein: Setzt *bSound* entsprechend dem übergebenen Parameter.
TDevice::~TDevice(), *TGenericDevice::Initialize()*, ***TCommonDevParam::Dlg_OnCommand()***, ***TCommonDevParam::LeaveDialog()***

SetSpezificParametersDlg
Allgemein: Öffnet eine Dialogbox mit detektorspezifischen Einstellungen. (Radicon SCSCS)
TSetupAreaScan::Dlg_OnCommand(), ***TCommonDevParam::Dlg_OnCommand()***

SingleStepMeasurement
Gibt 0 zurück. (War ursprünglich für *TMatrox*-Framegrabber gedacht, wird aber nie aufgerufen.)

UpdateDisplay
Allgemein: Sendet eine Nachricht (*cm_CounterSet*) an das GUI von Windows, das Zähler-Fenster zu aktualisieren.
TPsdRemoteSync::Dlg_OnCommand(), ***TCalibratePsd::Dlg_OnCommand()***, ***TMacroExecute::Dlg_OnCommand()***, ***TAdjustmentExecute::Dlg_OnCommand()***, ***TTopographyExecute::Dlg_OnCommand()***, ***TScan::CounterSetRequest()***, ***TAngleControl::Dlg_OnCommand()***

8. Fehler, Probleme

Design
Die Klasse ist eindeutig zu groß. 47 Funktionen, davon 44 mit öffentlichem Zugriff sind für die Wurzelklasse aller Zähler eindeutig zu viel, um sie vernünftig verstehen und handhaben zu können. Eine Aufteilung ist allemal angebracht.
Die Verwendung der Wurzelklasse aller Detektoren zur Implementation eines Testzählers ist mehr als fraglich. Sinnvollerweise hätte man TDevice zur abstrakten Klasse machen sollen, die für alle von ihr abgeleiteten Klassen ein Interface darstellt.
Alle Attribute sind (bis auf zwei Ausnahmen) mit öffentlichem Zugriff definiert. Damit ist die Forderung nach Datenkapselung nicht erfüllt und von einem Interface kann eigentlich nicht mehr gesprochen werden. Als Folge läßt sich nicht mehr klar eingrenzen, an welcher Stelle des Programmes die Attribute verändert werden. Da nun mögliche Fehler im gesamten Programm versteckt sein können, lassen sich Fehler sehr viel schwerer lokalisieren und der Wartungsaufwand steigt ins Unermeßliche.

Attribute und Methoden

bAnalysator

Die Funktion von bAnalysator ist nicht erkennbar. bAnalysator wird in TCommonDevParam:: Dlg_OnCommand verwandt, wo der Wert von bAnalysator höchstwahrscheinlich durch ein Kontrollkästchen geregelt werden sollte. Das Kontrollkästchen mit der ID id_Analysator existiert jedoch gar nicht im zugehörigen Dialog (counters.rc).

bDiskriminator

Die Funktion von bDiskriminator ist nicht erkennbar.

bStaticFailure

Das Attribut kann bisher nur in einem Dialogfenster eingestellt werden, wird aber sonst nicht verwandt.

CalcTime

..Sollte eine Klassenmethode werden.

dwLowerCountBound

dwLowerCountBound ist als Variable implementiert, ist inhaltlich aber eine Konstante, und sollte also auch als eine solche geführt werden.

dwUpperCountBound

dwUpperCountBound ist als Variable implementiert, ist inhaltlich aber eine Konstante, und sollte also auch als eine solche geführt werden.

fLowerTimeBound

fLowerTimeBound ist als Variable implementiert, ist inhaltlich aber eine Konstante, und sollte also auch als eine solche geführt werden.

fUpperTimeBound

fLowerTimeBound ist als Variable implementiert, ist inhaltlich aber eine Konstante, und sollte also auch als eine solche geführt werden.

GetMFunction

GetMFunction wird im gesamten Projekt nicht verwendet. Es stellt wahrscheinlich den rudimentären Ansatz dar, den aktuellen Arbeitsmodus eines Zählers über eine Interface-Funktion zu ermitteln.

MeasureStopExternal

Die Methode wird nie aufgerufen. Ihre Funktion ist nicht ableitbar. Es ist nicht erkennbar, warum MeasureStop nicht ausreichen sollte.

nFunctionId

nFunctionId kann zwar von je einer Interface-Methode gesetzt und gelesen werden, wird aber sonst nicht verwandt. (siehe GetMFunction bzw. SelectMFunction)

nPeakLevel

Die Funktion von nPeakLevel ist nicht erkennbar. Das Attribut wird auf 13% gesetzt und nie wieder verwandt.

nPeakWindow

Die Funktion von nPeakWindow ist nicht erkennbar. Das Attribut wird auf 15% gesetzt und nie wieder verwandt.

SelectMFunction

SelectMFunction wird im gesamten Projekt nicht verwendet. Es stellt wahrscheinlich den rudimentären Ansatz dar, die verschiedenen Arbeitsmodi der Zähler über eine Interface-Funktion zu setzen.

SingleStepMeasurement

SingleStepMeasurement ist in TDevice definiert und sollte offensichtlich in TMatrox verwandt werden, wird aber auch dort nicht implementiert.

class TAm9513a

1. UML-Klassendiagramm

class TAm9513a	
	- bIsLatched : BOOL - bIsStopped : BOOL - fTimeCorrection : float - nBaseAddr : int # nbChipId : BYTE - wHighCounts : WORD - wHighTicks : WORD - wLowCounts : WORD - wLowTicks : WORD
<<virtual>>	+ TAm9513a(void) + ~TAm9513a(void) # ArmC(BYTE) : void # ChooseDataPtr(BYTE group, BYTE reg) : void + ClearToggleOut(BYTE) : void + DigitalIn(BYTE*) : void + DigitalOut(BYTE) : void # DisarmAndSaveC(BYTE) : void # DisarmC(BYTE) : void + GetTicksPerSecond(void) : DWORD + Init(void) : int + IOCTL(TIOCCmd) : int + IOCTL(TIOCCmd, DWORD&) : int # LatchToHoldC(BYTE) : void # LoadAndArmC(BYTE) : void # LoadC(BYTE) : void + LoockUp(LPCSTR) : void # ReadData(void) : WORD # ReadStatus(void) : WORD # SelectChip(BYTE) : BOOL + SetToggleOut(BYTE) : void - SplitNumber(DWORD, WORD&, WORD&) : BOOL # WriteCmd(BYTE cmd) : void # WriteData(WORD) : void

2. Klassenhierarchie

TAm9513a ist von keiner Klasse abgeleitet.

3. Friends

class TGenericDevice
class TCCDScan
BOOL TriggerLH(void)
BOOL WINAPI GrabLineData(void)

4. Files

Klassendefinition: am9513a.h
Implementation: am9513a.cpp

5. Verantwortlichkeiten der Klasse

TAm9513a kapselt den Zugriff auf die Controllerkarte AXIOM AX5216 mit dem Controllerchip Am9513a.

6. Beschreibung der Attribute

bIsLatched

Ist wahrscheinlich eine Art Semaphor und soll den Controller zeitweilig für weitere Zugriffe durch die Software sperren.
TAm9513a::TAm9513a(), TAm9513a::IOCTL()

bIsStopped

Spiegelt wahrscheinlich den Status des Controllers wider, also ob gerade aktiv eine Messung durchgeführt wird oder nicht.
TAm9513a::TAm9513a(), TAm9513a::IOCTL()

fTimeCorrection

Korrekturwert für die Zeitbegrenzung (Vermutung) (wird aus der ini-Datei eingelesen)
TAm9513a::LoockUp(), *TAm9513a::IOCTL()*

nBaseAddr

I/O-Basisadresse für die Kommunikation mit dem Controller.
(wird aus der ini-Datei eingelesen *TAm9513a::LoockUp()*)
TAm9513a::LoockUp(), *TAm9513a::ReadStatus()*, *TAm9513a::WriteCmd()*, *TAm9513a::WriteData()*, *TAm9513a::ReadData()*

nbChipId

??? Wird in TAm9513a::TAm9513a() mit 0 geschrieben und in TAm9513a::WriteCmd() auf den Wert DIGITAL (02h) getestet und sonst nirgendwo geschrieben bzw. gelesen - scheint demnach keine sinnvolle Funktion zu besitzen.
TAm9513a::TAm9513a(), *TAm9513a::WriteCmd()*

wHighCounts

Höheres Word des Wertes für die Impulsbegrenzung.
TAm9513a::TAm9513a(), TAm9513a::IOCTL()

wHighTicks

Höheres Word des Wertes für die Zeitbegrenzung.
TAm9513a::TAm9513a(), TAm9513a::IOCTL()

wLowCounts

Niederes Word des Wertes für die Impulsbegrenzung.
TAm9513a::TAm9513a(), TAm9513a::IOCTL()

wLowTicks

Niederes Word des Wertes für die Zeitbegrenzung.
TAm9513a::TAm9513a(), TAm9513a::IOCTL()

7. Beschreibung der Methoden

TAm9513a
Konstruktor. Initialisiert einige Attribute (*bIsStopped, bIsLatched, wLowTicks, wHighTicks, wLowCounts, wHighCounts, nbChipId*)
(Wird nie aufgerufen.)

~TAm9513a
Destruktor. (nicht implementiert)

ArmC
Senden des Kommandos *0x20* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

ChooseDataPtr
Auswählen eines Registers des Controllers zur Kommunikation.
TAm9513a::Init(), *TAm9513a::IOCTL()*, *TAm9513a::SelectChip()*

ClearToggleOut
Senden des Kommandos *0xE0* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

DigitalIn
Liest ein Byte direkt von der I/O-Adresse (Port) *nBaseAddr+2*. (nur Win16) (Wird nie aufgerufen.)

DigitalOut
Schreibt ein Byte direkt an die I/O-Adresse (Port) *nBaseAddr+3*. (nur Win16)
TGenericDevice::SetSound()

DisarmAndSaveC
Senden des Kommandos *0x80* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
(Wird nie aufgerufen.)

DisarmC
Senden des Kommandos *0xC0* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

GetTicksPerSecond
Liefert lediglich den Wert *100000L* zurück.
TGenericDevice::SetParameters(), *TGenericDevice::PollDevice()*

Init
Konfiguration controllerspezifischer Eigenschaften durch Kommunikation mit der Controllerhardware.
TGenericDevice::Initialize()

IOCTL(TIOCCmd)
1-parametriges Interface zu *IOCTL(TIOCCmd, DWORD&)*. Dazu wird der erforderliche zweite Parameter auf 0 gesetzt.
TGenericDevice::MeasureStart(), *TGenericDevice::MeasureStop()*, *TGenericDevice::PollDevice()*

IOCTL(TIOCCmd, DWORD&)
Interpretiert komplexe Kommandos und setzt diese in Folgen von spezifischen Controllerkommandos um.
(*iocSetTimeTicks*, *iocSetCounts*, *iocStopCCDTiming*, *iocStartCCDTiming*, *iocStartCCDShot*, *iocReadTicks*, *iocReadCounts*, *iocConfigureAcustic*, *iocStartCounting*, *iocIsCountingReady*, *iocStopCounting*)
TAm9513a::IOCTL(), *TGenericDevice::Initialize()*, *TGenericDevice::SetParameters()*, *TGenericDevice::PollDevice()*

LatchToHoldC
Senden des Kommandos *0xA0* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

LoadAndArmC
Senden des Kommandos *0x60* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

LoadC
Senden des Kommandos *0x40* | *counter* an den Contoller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

LoockUp
nBaseAddr und *fTimeCorrection* werden mit Werten aus der ini-Datei initialisiert.
TGenericDevice::TGenericDevice()

ReadData
Einlesen eines Datenwortes vom Datenport des Controllers (2 x 1Byte). (nur Win16)
TAm9513a::Init(), *TAm9513a::IOCTL()*, *TAm9513a::SelectChip()*

ReadStatus
Einlesen der Detektorantworten am Kommandoport des Controllers. (nur Win16)
TAm9513a::IOCTL()

SelectChip
Testet die Kommunikation mit dem Controller ? (Wird nie aufgerufen.)

SetToggleOut
Senden des Kommandos *0xE8* | *counter* an den Controller, wobei *counter* der Methode übergeben wird.
TAm9513a::IOCTL()

SplitNumber
Zerlegt ein DWORD in 2 WORD (HighWord und LowWord).
TAm9513a::IOCTL()

WriteCmd
Schreiben eines Detektorkommandos an den Kommandoport des Controlllers. (nur Win16)
TAm9513a::Init(), *TAm9513a::SelectChip()*, *TAm9513a::ArmC()*, *TAm9513a::LoadC()*, *TAm9513a::LoadAndArmC()*, *TAm9513a::DisarmAndSaveC()*, *TAm9513a::LatchToHoldC()*, *TAm9513a::DisarmC()*, *TAm9513a::ClearToggleOut()*, *TAm9513a::SetToggleOut()*, *TAm9513a::ChooseDataPtr()*

WriteData
Senden eines Datenwortes an den Datenport des Controllers (2 x 1Byte). (nur Win16)
TAm9513a::Init(), *TAm9513a::IOCTL()*, *TAm9513a::SelectChip()*

8. Fehler, Probleme

Die fehlende Beschreibung der Register und der Kommandos des Am9513a erschwert das Verständnis des Quelltextes.

class TGenericDevice

1. UML-Klassendiagramm

class TGenericDevice	
	Siehe class TDevice und class TAm9513a
	+ TGenericDevice()
<<static>> <<CALLBACK>>	+ EventHandler(UINT, UINT, DWORD, DWORD, DWORD)
	+ Initialize(void) : int
	+ InitializeEvent(HWND, int) : BOOL
	# MeasureStart(void) : int
	# MeasureStop(void) : int
	# PollDevice(void) : int
	# SetParameters(void) : int
	# SetSound(void) : BOOL

2. Klassenhierarchie

TGenericDevice ist public von TDevice und TAm9513a abgeleitet.

3. Friends

keine...

4. Files

Klassendefinition: m_devhw.h
Implementation: counters.cpp

5. Verantwortlichkeiten der Klasse

TGenericDevice implementiert die Methoden für einen generischen Detektor, also für einen Detektor, der an der AXIOM AX5216 Schnittstellenkarte betrieben wird (z.B. russischer SCSCS).

6. Beschreibung der Attribute

Siehe class TDevice und class TAm9513a.

7. Beschreibung der Methoden

TGenericDevice

Konstruktor. Initialisiert einige Detektorattribute (*nBaseAddr*, *fTimeCorrection* und *uAskTime*)
TDList::InitializeModule()

EventHandler

Liefert lediglich den Returncode *R_MeasOk* zurück. (allgemeine Funktion siehe TDevice)

Initialize

Initialisiert und setzt einige Detektorattribute (u.a. Meß-Limits).
TDList::InitializeModule()

InitializeEvent

Liefert lediglich den Returncode *R_MeasOk* zurück. (allgemeine Funktion siehe TDevice)

MeasureStart

Stoppt zuerst eine evtl. laufende Messung und startet eine neue. Zusätzlich wird der Detektor als aktiv gesetzt und startet den Meß-Timer.

MeasureStop

Wenn eine Messung aktiv ist, werden der Detektor und der Meß-Timer gestoppt. Anschließend werden die vorliegenden Daten als ungültig und der Detektor als nicht aktiv markiert.
TGenericDevice::MeasureStart()

PollDevice

Testet, ob eine Messung läuft und bricht in diesem Fall mit einem Fehler ab. Sollte die Messung beendet sein, so werden die Meßergebnisse (*counts* und *time*) ausgelesen und daraus *fIntensity*, als gesuchter Wert ermittelt, die Daten als gültig und der Detektor als nicht aktiv markiert (⇒ siehe *TDevice::PollDevice()*) Abschließend wird ein Fenster (welches ???) mit der Nachricht *wpJumpTo* darüber informiert, daß jetzt gültige Daten vorliegen.

SetParameters

Konfiguriert den Detektor entsprechend den Attributen *fExposureTime* und *dwExposureCounts* (Meß-Limits).

SetSound

Konfiguriert den Detektor entsprechend dem Attribut *bSound* (akustisches Feedback).
TGenericDevice::Initialize()

8. Fehler, Probleme

Irgendwie ist die Aufgabenteilung bzw. der Unterschied zwischen *Initialize()* und *SetParameters()* nicht ganz zu klären.

class TEncoder

1. UML-Klassendiagramm

class TEncoder
- bAnnounceActive : BOOL - dRealTime : double
+ ~Tencoder() + Tencoder(void) + GetData(float &) : int + GetData(TCurve &) : int + GetData(WORD *, WORD, WORD) : int + Initialize(void) : int - PollDevice(void) : int

2. Klassenhierarchie

TEncoder ist public von TDevice abgeleitet.

3. Friends

keine

4. Files

Klassendefinition: m_devhw.h
Implementation: counters.cpp

5. Verantwortlichkeiten der Klasse

Implementiert Methoden für die Geräteklasse Encoder.

6. Beschreibung der Attribute

bAnnounceActiv
Wird lediglich definiert, jedoch nie wieder verwendet.

dRealTime
Wird lediglich definiert, jedoch nie wieder verwendet.

7. Beschreibung der Methoden

TEncoder
Ruft lediglich den Konstruktor von *TDevice* auf.
TDList::InitializeModule()

~TEncoder
Der Destruktor ist leer.
TDList::~TDList()

GetData (float &)

Wenn keine gültigen Daten vorliegen, endet die Methode mit Fehler, andernfalls wird der Wert von *fIntensity* zurückgegeben (Zurückgeben des Meßwertes).
Siehe TDevice::GetData(float &)

GetData(TCurve &)

Gibt 0 zurück.
Siehe TDevice::GetData(TCurve &)

GetData(WORD *, WORD, WORD)

Gibt 0 zurück. (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

Initialize

Ruft zuerst *Initialize()* von *TDevice* auf. *bDeviceOk* wird auf TRUE gesetzt. *fExposureTime* wird konstant auf *0.2* gesetzt und damit der in *TDevice::Initialize()* aus dem ini-File gelesene Wert überschrieben. Der Hardwaretyp *eHardware* wird als *EncoderHW* festgelegt. Nun wird abhängig von *bDeviceOk* der Rückgabewert ermittelt.
TDList::InitializeModule()

PollDevice

Endet sofort mit Fehler, wenn noch eine Messung aktiv ist. Dann wird der Meß-Timer gestoppt. Mit den m_layer-Funktionen *mlIsAxisValid()* und *mlGetDistance()* wird ein Meßwert ermittelt und nach *fIntensity* geschrieben. Nun wird das Gerät als inaktiv, die Meßdaten als gültig markiert.

8. Fehler, Probleme

Welche Art von Gerät soll durch diese Klasse gekapselt werden?

bAnnounceActiv

bAnnounceActiv wird lediglich definiert, jedoch nie wieder verwendet.

dRealTime

dRealTime wird lediglich definiert, jedoch nie wieder verwendet.

Initialize

Der Rückgabewert der Methode wird in Abhängigkeit von *bDeviceOk* ermittelt, obwohl bDeviceOk zwei Zeilen vorher konstant auf TRUE gesetzt wurde.

class TRadicon

1. UML-Klassendiagramm

class TRadicon		
	#	bAnnounceActive : BOOL
	#	dRealTime : double
<<static>>	#	Rcc : int
<<static>>	#	Rdd : int
	#	wDacLowerThresh : WORD
	#	wDacUpperThresh : WORD
	#	wHighVoltage : WORD
<<static>>	+	EventHandler(UINT, UINT, DWORD, DWORD, DWORD)
	#	FailureOccured(int, int) : BOOL
	+	Initialize(void) : int
	+	InitializeEvent(HWND, int) : BOOL
	#	MeasureStart(void) : int
	#	MeasureStop(void) : int
	#	PollDevice(void) : int
	#	PSound(int, long) : BOOL
	#	SetParameters(void) : int
	#	SetSound(void) : BOOL
	#	SetSpecificParametersDialog(void)
	+	TRadicon
	+	~TRadicon

2. Klassenhierarchie

TRadicon ist public von TDevice abgeleitet.

3. Friends

class TScsParameters

4. Files

Klassendefinition: m_devhw.h
Implementation: counters.cpp

5. Verantwortlichkeiten der Klasse

TRadicon implementiert den Zugriff auf den Radicon-Szintilisationszähler. Wichtigste Aufgabe der Klasse ist die Kapselung des Zugriffs auf die Hardware mittels der low- und mid-Level-Treiber, die für den Radicon-Zähler in C implementiert sind.

6. Beschreibung der Attribute

bAnnounceActive

Die Bedeutung ist unklar, es wird lediglich einmal gesetzt, jedoch nie gelesen.
TRadicon::TRadicon()

dRealTime

Beinhaltet die tatsächlich gemessene Zeit.
TRadicon::MeasureStart(), *TRadicon::PollDevice()*, ***TAreaScan::CounterSetRequest()***

Rcc

Beinhaltet die Adresse des Steuerports der Radicon-Controllerkarte.
TRadicon::TRadicon(), *TRadicon::Initialize()*, *TRadicon::MeasureStart()*, *TRadicon::MeasureStop()*, *TRadicon::PollDevice()*, *TRadicon::SetParameters()*, *TRadicon::EventHandler()*, *TRadicon::InitializeEvent()*

Rdd

Beinhaltet die Adresse des Datenports der Radicon-Controllerkarte.
TRadicon::TRadicon(), *TRadicon::Initialize()*, *TRadicon::MeasureStart()*, *TRadicon::MeasureStop()*, *TRadicon::PollDevice()*, *TRadicon::SetParameters()*, *TRadicon::EventHandler()*, *TRadicon::InitializeEvent*

wDacLowerThresh

Beinhaltet die untere Schranke für die Hochspannung, mit der das Gerät betrieben wird.
TRadicon::TRadicon(), *TRadicon::~TRadicon()*, *TRadicon::SetParameters()*, *TScsParameters::Dlg_OnCommand()*, *TScsParameters::CanClose()*

wDacUpperThresh

Beinhaltet die obere Schranke für die Hochspannung, mit der das Gerät betrieben wird.
TRadicon::TRadicon(), *TRadicon::~TRadicon()*, *TRadicon::SetParameters()*, *TScsParameters::Dlg_OnCommand()*, *TScsParameters::CanClose()*

wHighVoltage

Beinhaltet die aktuelle Betriebsspannung des Gerätes.
TRadicon::TRadicon(), *TRadicon::~TRadicon()*, *TRadicon::SetParameters()*, *TScsParameters::Dlg_OnCommand()*, *TScsParameters::CanClose()*

7. Beschreibung der Methoden

TRadicon

Der Konstruktor liest die Radicon-spezifischen Einstellungen aus dem ini-File und initialisiert damit die Attribute.
TDList::InitializeModule()

~TRadicon

Der Destruktor schreibt die aktuellen Einstellungen für die untere und die obere Spannungsschranke, die aktuelle Betriebsspannung und die Sound-Einstellungen zurück in das ini-File.
TDList::~TDList()

EventHandler

Liest die Meßzeit und den Meßwert aus dem Radicon und berechnet daraus *fEventIntensity*. Wenn die Methode *nEventCalls*–mal ausgeführt wurde, wird der zugehörige Event-Timer gekillt, die Nachricht *cm_SteeringReady* abgesetzt und die (Event!)-Daten als ungültig markiert.
Wurde die Methode noch nicht *nEventCalls* –mal ausgeführt, so wird nur die Nachricht *cm_CounterSet* abgesetzt.

FailureOccured

Wertet die Fehlermeldungen vom Radicon-Controller aus und gibt entsprechend eine Fehlermeldung in einer Message-Box aus.
TRadicon::MeasureStart(), *TRadicon::MeasureStop()*, *TRadicon::PollDevice()*, *TRadicon::SetParameters()*, *TRadicon::InitializeEvent()*

Initialize

Lädt das zur Kommunikation mit Detektor notwendige Programm *scs.prg* und konfiguriert den Detektor mit Initialisierungswerten.
TDList::InitializeModule()

InitializeEvent

Ermittelt zuerst ein Intervall aus *fExposureTime*, und prüft, ob es groß genug ist. Die Meßdaten werden als ungültig markiert. Wenn das Gerät aktiv ist, wird es als inaktiv markiert und das Gerät angehalten. Dann wird der Radicon im Intensimeter-Modus gestartet, das Gerät als aktiv und die Meßdaten als gültig markiert. Dann wird ein *timeSetEvent()*-Ereignistimer gestartet, der periodisch *TRadicon::EventHandler* aktiviert.
TScan::InitializeTask()

MeasureStart

Stoppt zuerst eine eventuell noch laufende Messung mit *MeasureStop()*. Dann wird das Gerät im Time-Count-Modus gestartet. Sollte die Messung vor Ablauf der Zeit beendet worden sein, z.B. weil die maximale Impulszahl erreicht worden ist, so wird der Zählwert aus gemessenen Impulsen und der verstrichenen Meßzeit errechnet. Das Gerät wird dann als inaktiv und die Daten als gültig markiert. Andernfalls wird das Gerät als aktiv und die Daten als ungültig markiert, und ein Meß-Timer gestartet.
Siehe TDevice::MeasureStart()

MeasureStop

Kehrt sofort zurück, wenn keine Messung aktiv ist. Andernfalls wird der Meß-Timer gekillt, das Gerät gestoppt und als inaktiv markiert. Die Meßdaten sind zu diesem Zeitpunkt ungültig. Anschließend wird gewartet, bis die Hardware reagieren konnte.
TRadicon::MeasureStart()

PollDevice

Killt zuerst den Meß-Timer. Anschließend wird das Gerät ausgelesen und die Meßwerte entsprechend der gemessenen Impulsanzahl und der verstrichenen Meßzeit berechnet. Die Daten sind nun gültig und das Gerät inaktiv. Anschließend wird eine *wpJumpTo*-Nachricht abgesetzt.
Siehe TDevice::PollDevice()

PSound

Ist vorsichtshalber nicht implementiert.

SetParameters

Schreibt die Attributwerte für die Meßzeit, die obere und untere Schranke für die Hochspannung, die Spannung und den Sound hardwaremäßig in das Gerät.
TRadicon::Initialize(), *TRadicon::SetSound()*

SetSound

Ruft lediglich *SetParameters()* auf.
Siehe TDevice::SetSound()

SetSpecificParametersDialog

Generiert einen neuen *ScsParametersDialog* und führt ihn aus.
Siehe TDevice::SetSpecificParametersDialog()

8. Fehler, Probleme

bAnnounceActive

Die Bedeutung von bAnnounceActive ist unklar, es wird lediglich einmal gesetzt, jedoch nie wieder verwendet.

InitializeEvent

InitializeEvent markiert die Meßdaten als gültig, obwohl zu diesem Zeitpunkt gar keine Daten ermittelt wurden. Es existiert ein auskommentierter Bereich, der den Radicon im Time-Count-Modus starten würde.

PSound

PSound ist vorsichtshalber nicht implementiert.

class TPsd

1. UML-Klassendiagramm

class TRadicon	
	# bFirstReadOut : BOOL # bHardOverflow : BOOL # bHVRegelung : BOOL # bReadLeftFirst : BOOL # bReadyForRead : BOOL # bSignalGrowUp : BOOL - bSoftOverflow : BOOL # dwIntegratedCounts : DWORD # dwMaxCounts : DWORD # eUnit : TUnitType - fAngleStep : float - fAngleWidth : float # hCountBuf : HGLOBAL # lSeconds : long # lWaitTime : long - nAddedChannels : int # nBaseAddr : int # nChannelNumber : int # nCountBufItems : int # nMaxChannel : int # nMinChannel : int - OverflowIntensity : float - s_AngleStep : float - s_nAddedChannels : int # szUnit[] : char # uMaximumChannel : UINT
<<virtual>>	+ ~TPsd(void) + GetAddedChannels(void) : int + GetAngleRange(TUnitType) : float + GetAngleRange(void) : float + GetAngleStep(void) : float + GetBufferSize(void) : int + GetChannelNumber(void) : int + GetData(float&) : int + GetData(TCurve&) : int + GetData(WORD*, WORD, WORD) : int + GetEnergyRange(UINT&, UINT&) : int + GetMaximumChannel(void) : UINT + GetMaximumIntensity(void) : float + GetUnitStr(void) : LPCSTR + GetWidth(TUnitType) : float + GetWidth(void) : float + Initialize(void) : int + IsHardOverflow(void) : BOOL + IsSoftOverflow(void) : BOOL + MeasureStart(void) : int + MeasureStop(void) : int

	+	MeasureStopExternal(void) : int
<<virtual>>	+	PollDevice(void) : int
	+	PopSettings(void) : BOOL
<<virtual>>	-	PsdInit(void) : int
<<virtual>>	-	PsdRead(int, int, int, LPWORD) : int
<<virtual>>	-	PsdReadOut(THowReadOutPsd) : int
<<virtual>>	-	PsdStart(void) : int
<<virtual>>	-	PsdStop(void) : int
	+	PushSettings(void) : BOOL
	+	SetAddedChannels(int)
	+	SetAngleRange(float)
	+	SetAngleStep(float)
<<virtual>>	+	SetEnergyRange(UINT, UINT) : int
	+	SetParameters(void) : int
	+	SetSignalGrowUp(BOOL)
	+	SetSpezificParametersDlg( );
	+	TPsd(void)

2. Klassenhierarchie

TPsd ist public von TDevice abgeleitet.

3. Friends

class TPsdParameters

4. Files

Klassendefinition: m_psd.h
Implementation: counters.cpp

5. Verantwortlichkeiten der Klasse

TPsd definiert das Interface für die eindimensionalen Detektoren vom Typ P(ositional) S(ensitive) D(evice).

6. Beschreibung der Attribute

bFirstReadOut

Status-Flag, das anzeigt, daß der Psd in diesem Meßzyklus das erste Mal ausgelesen wird.
TPsd::MeasureStart(), *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*

bHardOverflow

Status-Flag, das anzeigt, daß im Gerät ein Überlauf stattgefunden hat.
TPsd::TPsd(), *TPsd::GetData()*, *TPsd::MeasureStart()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TStoe_Psd::PsdStop()*, *TStoe_Psd::PsdRead()*, *TBraun_Psd::PsdReadOut()*, *TPsd::IsHardOverflow()*

bHVRegelung

Status-Flag, das gesetzt wird, wenn die Hochspannung am Gerät gesteuert werden soll.
TPsd::TPsd(), *TPsd::~TPsd()*, *TPsd::Initialize()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::PsdInit()*, *TPsdParameters::Dlg_OnCommand()*, *TPsdParameters::CanClose()*

bReadLeftFirst
Status-Flag, das gesetzt wird, wenn beim Auslesen der Meßwerte aus dem Gerät von links begonnen werden soll.
TPsd::Initialize(), *TPsd::GetMaximumChannel()*, *TPsd::GetData()*, *TPsdParameters::Dlg_OnCommand()*, *TPsdParameters::CanClose()*

bReadyForRead
Entspricht dem Gegenteil des Status-Flags *bDeviceActive*. Es ist dann gesetzt, wenn der Meßvorgang abgeschlossen ist und die Daten jetzt ausgelesen werden können.
TStoe_Psd::PsdInit(), *TStoe_Psd::PsdStart()*, *TStoe_Psd::PsdStop()*, *TStoe_Psd::PsdRead()*

bSignalGrowUp
Status-Flag, das gesetzt wird, wenn die Meßdaten nicht nur am Ende des Meßvorgangs ausgelesen werden sollen, sondern auch Zwischenablesungen stattfinden sollen.
TPsd::TPsd(), *TPsd::Initialize()*, *TPsd::PollDevice()*, *TStoe_Psd::PollDevice()*, *TPsd::SetSignalGrowUp()*

bSoftOverflow
Status-Flag, das gesetzt wird, wenn während einer Messung ein Meßwert aus gemessener Intensität und verstrichener Zeit berechnet worden ist, der größer als *OverflowIntensity* ist.
TPsd::TPsd(), *TPsd::GetData()*, *TPsd::MeasureStart()*, *TPsd::IsSoftOverflow()*

dwIntegratedCounts
Beinhaltet die Summe der Meßwerte der einzelnen Kanäle.
TPsd::GetData(), *TPsd::PsdReadOut()*, *TStoe_Psd::PsdReadOut()*, *TBraun_Psd::PsdReadOut()*

dwMaxCounts
Beinhaltet die größte Einzelintensität, die auf einem der Kanäle gemessen worden ist.
TPsd::PsdReadOut(), *TStoe_Psd::PsdReadOut()*, *TBraun_Psd::PsdReadOut()*, *TPsd::GetMaximumIntensity()*

eUnit
Beinhaltet die Einheit, in der Winkelmaße geführt werden.
TPsd::Initialize(), *TPsd::SetAddedChannels()*, *TPsd::GetAngleRange()*

fAngleStep
Beinhaltet den Winkel, der von einem Kanal abgedeckt wird.
TPsd::Initialize(), *TPsd::GetWidth()*, *TPsd::SetAddedChannels()*, *TPsd::GetAngleRange()*, *TPsd::SetAngleStep()*, *TPsd::GetAngleStep()*, *TPsdParameters::Dlg_OnCommand()*

fAngleWidth
Beinhaltet den Winkel, der von einer Kanalgruppe abgedeckt wird.
TPsd::Initialize(), *TPsd::SetAddedChannels()*, *TPsd::GetWidth()*

hCountBuf
Puffer, welcher der Meßwertübernahme aus der Hardware in das Objekt dient.
TPsd::Initialize(), *TPsd::GetData()*, *TPsd::PsdReadOut()*, *TStoe_Psd::PsdReadOut()*, *TBraun_Psd::PsdReadOut()*

lSeconds
lSeconds wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

lWaitTime
Beinhaltet die Zeit, die noch zu messen ist.
TPsd::PollDevice(), *TStoe_Psd::PollDevice()*

nAddedChannels

Beinhaltet die Anzahl der Kanäle, deren Meßwerte zu einem gemeinsamen Meßwert zusammengefaßt werden.
TPsd::~TPsd(), *TPsd::GetWidth()*, *TPsd::SetAddedChannels()*, *TPsd::GetData()*, *TPsd::PushSettings()*, *TPsd::PopSettings()*, *TPsd::GetAddedChannels()*, *TPsdParameters::Dlg_OnCommand()*

nBaseAddr

Beinhaltet die IO-Adresse für die Kommunikation mit der Hardware.
TPsd::Initialize(), *TStoe_Psd::PsdInit()*, *TStoe_Psd::PsdStart()*, *TStoe_Psd::PsdStop()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::BuildOperation()*, *TBraun_Psd::ResetDelayTime()*, *TBraun_Psd::Look_till_BaseAddr1()*, *TBraun_Psd::SynchronHexFile()*, *TBraun_Psd::LoadHexFile()*

nChannelNumber

Beinhaltet die Anzahl der Kanäle.
TPsd::Initialize(), *TBraun_Psd::PsdReadOut()*, *TPsd::GetBufferSize()*, *TPsd::GetChannelNumber()*

nCountBufItems

nCountBufItems wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

nMaxChannel

Beinhaltet die Nummer des letzten zu benutzenden Kanals.
TPsd::~TPsd(), *TPsd::Initialize()*, *TStoe_Psd::PsdReadOut()*, *TBraun_Psd::PsdReadOut()*, *TStoe_Psd::GetBufferSize()*, *TPsdParameters::Dlg_OnCommand()*

nMinChannel

Beinhaltet die Nummer des ersten zu benutzenden Kanals.
TPsd::~TPsd(), *TPsd::Initialize()*, *TStoe_Psd::PsdReadOut()*, *TBraun_Psd::PsdReadOut()*, *TStoe_Psd::GetBufferSize()*, *TPsdParameters::Dlg_OnCommand()*

OverflowIntensity

Beinhaltet den maximal möglichen Meßwert.
TPsd::Initialize(), *TPsd::GetData()*

szUnit[]

Beinhaltet die Einheit, in der Winkel angegeben werden, im Klartext.
TPsd::Initialize(), *TPsd::GetUnitStr()*

s_AngleStep

s_AngleStep wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

s_nAddedChannels

s_nAddedChannels wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

uMaximumChannel

Beinhaltet den Kanal, bei dem der größte Meßwert gemessen wurde.
TPsd::GetMaximumChannel(), *TPsd::PsdReadOut()*, *TStoe_Psd::PsdReadOut()*, *TBraun_Psd::PsdReadOut()*

7. Beschreibung der Methoden

TPsd

Konstruktor. Ruft zuerst den Konstruktor von *TDevice* auf. Dann werden die Attribute *bSoftOverflow*, *bHardOverflow*, *bSignalGrowUp* und *bHVRegelung* auf FALSE gesetzt.
TDList::InitializeModule(),*TBraun_Psd::Initialize()*

~TPsd

Destruktor. Schreibt die Einstellungen für *bHVRegelung*, *nMinChannel*, *nMaxChannel* und *nAddedChannels* in das ini-File.
TDList::~TDList()

GetAddedChannels

Gibt nAddedChannels zurück.
TAreaScanParameters::TAreaScanParameters(),TSetupAreaScan::Dlg_OnCommand()

GetAngleRange(TUnitType eunit)

Berechnet den gemessenen Winkelbereich. Abhängig vom Parameter *eunit* wird *fAngleStep* zuerst in Bogensekunden umgerechnet und mit der Anzahl der Kanäle multipliziert.
TPsd::GetAngleRange(), ***TAreaScanParameters::TAreaScanParameters(), TAreaScan::SetRanges(), TAreaScan::InitializeTask()***

GetAngleRange(void)

Ruft lediglich *GetAngleRange(TUnitType eunit)* mit dem richtigen Parameter auf.
TPsdParameters::Dlg_OnCommand(),***TAreaScan::SetRanges***

GetAngleStep

Gibt *fAngleStep* zurück.
TPsd::PsdReadOut(), ***TAreaScan::SetRanges()***, ***TCalibratePsd::Dlg_OnCommand()***, ***TCalibratePsd::CanClose()***

GetBufferSize

Gibt (*nChannelNumber* + 1) zurück (Größe des Meßwertpuffers *hCountBuf*).
TPsd::Initialize(), *TBraun_Psd::TBraun_Psd()*

GetChannelNumber

Gibt nChannelNumber zurück.
TPsd::GetMaximumChannel(), TPsd::GetData(), TPsd::GetAngleRange(), TPsd::PsdReadOut(), TStoe_Psd::Initialize(), TStoe_Psd::PsdReadOut(), TPsdParameters::CanClose()

GetData(float &)

GetData(float &integral) endet mit Fehler, wenn ein Hardoverflow aufgetreten ist, oder die Zeit gleich 0 ist. *fIntensity* wird aus *dwIntegratedCounts* und der verstrichenen Zeit ermittelt. Sollte *fIntensity* nun größer als *OverflowIntensity* sein, gibt die Methode das als Fehler zurück. Siehe TDevice::GetData(float &)

GetData(TCurve &)

GetData(TCurve &curve) überprüft zuerst, ob ein Hardoverflow stattgefunden hat, wenn ja, endet die Methode mit Fehler. Der Puffer *hCountBuf* wird gesperrt. Wenn *dRealTime* 0 ist, wird mit Fehler beendet. Auf den Parameter *curve* werden *New()* und *FastOpen()* angewandt. Abhängig davon, ob zuerst von links oder von rechts gelesen werden soll, wird nun für die einzelnen Kanäle die Intensität ermittelt und ein Punkt in die Kurve eingefügt. Nun wird die Kurve abgeschlossen und das Lock auf *hCountBuf* entfernt. *fIntensity* wird aus *dwIntegratedCounts* und der verstrichenen Zeit ermittelt. Sollte *fIntensity* nun größer als *OverflowIntensity* sein, gibt die Methode das als Fehler zurück.
Siehe TDevice::GetData(TCurev &)

GetData(WORD*, WORD, WORD)

Gibt lediglich 0 zurück. (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

GetEnergyRange(UINT&, UINT&)

Setzt den ersten Parameter auf 1 und den zweiten auf 0.
TPsdParameters::Dlg_OnCommand()

GetMaximumChannel

Gibt die maximale Anzahl der Kanäle zurück.
Siehe TDevice::GetMaximumChannel()

GetMaximumIntensity

Berechnet entsprechend der tatsächlichen Meßzeit die Maximalintensität und gibt diese zurück.
Siehe TDevice::GetMaximumIntensity()

GetUnitStr

Gibt *szUnit* zurück. (Wird nie aufgerufen.)

GetWidth(TUnitType)

Gibt den Winkel der von einer Kanalgruppe abgedeckt wird (Produkt aus *nAddedChannels* und *fAngleStep*) in Sekunden zurück. Wenn nötig, wird über den Parameter *unit* die korrekte Einheit für *fAngleStep* verwandt.
TPsd::GetData()

GetWidth(void)

Gibt den Winkel der von einer Kanalgruppe abgedeckt wird (*fAngleWidth*) zurück.
Siehe TDevice::GetWidth()

Initialize

Initialisierung der Detektorumgebung. Ruft zuerst *TDevice::Initilialize()* auf. Dann werden PSD-spezifische Attribute mit Werten aus dem ini-File belegt. *nChannelNumber* wird aus den eingelesenen Werten ermittelt. Nun wird der Puffer *hCountBuf* von (*nAddedChannels+1*) DWORDs alloziiert. Jetzt wird *PsdInit()* aufgerufen, das Attribut *eHardware* auf *GenericPsdHW* gesetzt und *PushSettings()* aufgerufen.
TBraun_Psd::Initialize(), *TStoe_Psd::Initialize()*

IsHardOverflow

Gibt *bHardOverflow* zurück.
Siehe TDevice::IsHardOverflow()

IsSoftOverflow

Gibt *bSoftOverflow* zurück.
Siehe TDevice::IsSoftOverflow()

MeasureStart

MeasureStart beendet zuerst eine eventuell noch laufende Messung. Soft- und Hard Overflow werden auf FALSE gesetzt, bFirstReadOut auf TRUE. Nun wird *PsdStart()* ausgeführt, und *MeasureStart()* mit Fehler beendet, wenn *PsdStart()* einen Fehler zurückliefert.
Die Meßzeit wird aus *fExposureTime* berechnet, und ein Timer gestartet. Die Startzeit wird in *dwStartTime* festgehalten. Das Gerät wird nun als aktiv markiert und es wird festgelegt, daß jetzt keine gültigen Daten vorliegen können.
Siehe TDevice::MeasureStart()

MeasureStop

MeasureStop überprüft zuerst, ob die Messung nicht schon gestoppt ist. Wenn nicht, wird der Meß-Timer gekillt. Nun wird *PsdStop()* ausgeführt und das Gerät als inaktiv markiert.
TPsd::MeasureStart*()*

MeasureStopExternal

MeasureStopExternal berechnet die vergangene Zeit aus der aktuellen Zeit und der Startzeit der Messung. (Wird nie aufgerufen – scheint demnach keine relevante Funktion zu besitzen.)

PollDevice

PollDevice prüft zuerst, ob das Gerät als ok markiert ist(wenn nicht, wird mit Fehler beendet). Der Meß-Timer wird gestoppt. Ist das Gerät nicht aktiv, wird mit Fehler beendet. Jetzt wird die seit dem Start der Messung vergangene Zeit ermittelt und mit *fExposureTime* verglichen. Wurde nicht so lange gemessen, wie *fExposureTime* vorgibt, so wird das Gerät zuerst als aktiv markiert. Sollte stufenweiser Signalaufbau nicht gesetzt sein (*bSignalGrowUp*), wird der Meß-Timer wieder neu gestartet und mit *R_MeasureInProcess* beendet. Andernfalls wird mittels *PsdReadOut()* eine Zwischenauslesung der Meßwerte duchgeführt, die verbleibende Meßzeit neu berechnet, der Meß-Timer mit der verbleibenden Zeit gestartet und die Messung somit fortgeführt.
Ist *fExposureTime* abgelaufen, ist der Meßvorgang in diesem Intervall abgeschlossen. Die Daten werden als gültig markiert und *PsdStop()* aufgerufen. Nun wird die tatsächliche Meßzeit ermittelt und die Meßdaten ein letztes Mal mittels *PsdReadOut()* ausgelesen. Anschließend werden die Daten schon wieder als gültig markiert.
Jetzt werden Fehlerbedingungen geprüft: Ist die gemessene Zeit gleich 0, wird das gemeldet, gewartet, die Daten wieder als ungültig markiert und mit Fehler beendet. Hat ein Hard Overflow stattgefunden, so wird das akustisch mitgeteilt und der Rückkehrcode auf Fehler gesetzt.
Zum Ende wird ein *wpJumpTo* an das Fenster gesendet.
Siehe TDevice::PollDevice()

PopSettings

Liest Detektoreinstellung aus der *TDSettings*-Struktur.
Siehe TDevice::PopSettings()

PsdInit

Gibt lediglich *R_OK* zurück. (Wird von *TBraun_Psd* und *TStoe_Psd* implementiert.)
TPsd::Initialize(), *TPsdParameters::Dlg_OnCommand()*

PsdRead(int, int, int, LPWORD)

Gibt lediglich *R_OK* zurück. (Wird von *TStoe_Psd* implementiert.)
TStoe_Psd::PsdReadOut()

PsdReadOut

Ermittelt für die einzelnen Kanäle einen Meßwert, der aus einigen Konstanten und einem Zufallswert besteht. Offensichtlich handelt es sich hier um einen Psd-Testzähler.
TPsd::PollDevice(), TStoe_Psd::PollDevice()

PsdStart
Gibt lediglich *R_OK* zurück. (Wird von *TBraun_Psd* und *TStoe_Psd* implementiert.)
TPsd::MeasureStart()

PsdStop
Gibt lediglich *R_OK* zurück. (Wird von *TBraun_Psd* und *TStoe_Psd* implementiert.)
TPsd::MeasureStop(), *TPsd::PollDevice()*

PushSettings
Schreibt aktuelle Einstellungen in die *TDSettings*-Struktur.
Siehe TDevice::PushSettings()

SetAddedChannels
Setzt das Attribut *nAddedChannels* und berechnet *fAngleWidth*.
Siehe TDevice::SetAddedChannels()

SetAngleRange
SetAngleRange tut gar nichts. (Wird nie auferufen – scheint demnach keine relevante Funktion zu besitzen.)

SetAngleStep
Setzt das Attribut *fAngleStep*.
Siehe TDevice::SetAngleStep()

SetEnergyRange(UINT, UINT)
Gibt lediglich *R_OK* zurück. (Wird von *TBraun_Psd* implementiert.)
TPsdParameters::Dlg_OnCommand(), *TPsdParameters::CanClose()*

SetParameters
Gibt einfach lediglich TRUE zurück.
Siehe TDevice::SetParameters()

SetSignalGrowUp(BOOL)
Setzt das Attribut *bSignalGrowUp*.
Siehe TDevice::SetSignalGrowUp()

SetSpezificParametersDlg
SetSpezificParametersDlg tut gar nichts.
Siehe TDevice::SetSpecificParametersDlg()

8. Fehler, Probleme

GetData(TCurve &curve)
Wenn die Methode mit Fehler endet, weil *dRealtime==0* ist, wird das Lock auf *hCountBuf* nicht wieder entfernt. Das läßt sich einfach beheben, indem der dRealtime-Test direkt nach dem HardOverflow-Test durchgeführt wird, und erst anschließend gelockt wird.

Initialize
Aus noch unklaren Gründen gibt es einen Abschnitt, in dem die Reihenfolge der Anweisungen nicht verändert werden darf.

lWaitTime
lWaitTime ist zwar als Attribut definiert, ist aber lediglich eine Hilfsvariable, die in PollDevice verwandt wird. Sinnvollerweise sollte man also eine lokale Variable daraus machen.

MeasureStart

Die Bedeutung dieses Codefragmentes ist unklar.

```
if(!hControlWnd && !hDisplayWnd)
      {
      // MessageBox(GetFocus(),"Kein Host.","Startfehler",MBSTOP);
      return R_Failure;
      }
```

MeasureStop

bDataValid wird explizit auf FALSE gesetzt, obwohl es nur FALSE sein kann. (eventuell sollte es ja TRUE heißen?)

MeasureStopExternal

gibt in jedem Fall 1 zurück.

nCountBufItems, lSeconds, s_AngleStep, s_nAddedChannels

Die Attribute werden nicht verwendet, und können ohne Probleme entfernt werden.

PollDevice

bDeviceActive wird auf FALSE gesetzt, obwohl das Programm nie zu der Stelle gekommen wäre, wenn *bDeviceActiv*e einen anderen Wert als FALSE gehabt hätte. *retval* wird auf *R_MeasInProcess* gesetzt, um einige Zeilen später mit einem neuen Wert belegt zu werden.
Die Daten sind schon als gültig markiert, bevor das Gerät ein letztes Mal ausgelesen wurde. Anschließend werden sie nochmals auf gültig gesetzt.

PsdReadOut

*nNumbe*r wird am Anfang als *int* deklariert und mit *GetChannelNumber()* initialisiert. Anschließend wird es nicht weiter benutzt. Die Idee dabei war wahrscheinlich, daß die Anzahl der Kanäle in der Methode einmal ermittelt werden, und dann nur noch aus der Variable benutzt werden sollte. In der Realisierung wird *GetChannelNumber()* jedoch innerhalb jedes Schleifendurchlaufes zweimal aufgerufen. Die Methode verwendet den übergebenen Parameter überhaupt nicht. Im (*dRealtime==0*)-Test wird erst über den Fehler informiert und gewartet, um anschließend die Daten als ungültig zu markieren. Das sollte an sich zuerst geschehen. Im HardOverflow-Test wird eine Fehlermeldung in einen lokalen Puffer geschrieben, mit dem anschließend nichts passiert.

class TStoe_Psd

1. UML-Klassendiagramm

class TStoe_Psd
- bLong : BOOL - hReadBuf : HGLOBAL - nReadBufItems : int
+ GetBufferSize(void) : int - Initialize(void) : int + PollDevice(void) : int - PsdInit(void) : int - PsdRead(int, int, int, LPWORD) : int - PsdReadOut(THowReadOutPsd) : int - PsdStart(void) : int - PsdStop(void) : int - SetParameters(void) : int + TStoe_Psd(void) + ~TStoe_Psd(void)

2. Klassenhierarchie

TStoe_Psd ist public abgeleitet von TPsd.

3. Friends

keine...

4. Files

Klassendefinition: m_psd.h
Implementation: counters.cpp

5. Verantwortlichkeiten der Klasse

Kapselt den Zugriff auf die Hardware des Detektors StoePSD.

6. Beschreibung der Attribute

bLong
Beeinflußt, ob die Meßdaten als *int* bzw. als *long int* gespeichert werden.
TStoe_Psd::TStoe_Psd(), *TStoe_Psd::PsdReadOut()*

hReadBuf
Zeiger auf den Meßdatenpuffer.
TStoe_Psd::TStoe_Psd(), *TStoe_Psd::~TStoe_Psd()*, *TStoe_Psd::Initialize()*, *TStoe_Psd::PsdReadOut()*,

nReadBufItems

nReadBuffItems wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

7. Beschreibung der Methoden

TStoe_Psd

Konstruktor. Initialisiert die Attribute *bLong*, *hReadBuf* und *eHardware*.
TDList::InitializeModule()

~TStoe_Psd

Destruktor. Gibt den für den Meßdatenpuffer allokierten Speicher wieder frei.
TDList::~TDList()

GetBufferSize

Gibt *nMaxChannel-nMinChannel+1* (Anzahl der Kanäle+1) zurück.
Siehe TPsd::GetBufferSize()

Initialize

Ruft *TPsd::Initialize()* auf, allokiert Speicher für den Meßdatenpuffer (*hReadBuf*) und setzt das *TDevice*-Attribut *eHardware* auf *StoeHW*.
TDList::InitializeModule()

PollDevice

Überprüft zuerst, ob das Gerät als OK markiert ist (wenn nicht, wird mit Fehler beendet). Der Meß-Timer wird gestoppt. Ist das Gerät nicht aktiv, wird mit Fehler beendet. Jetzt wird die seit dem Start der Messung vergangene Zeit ermittelt und mit *fExposureTime* verglichen. Wurde nicht so lange gemessen, wie *fExposureTime* vorgibt, so wird das Gerät zuerst als aktiv markiert. Sollte stufenweiser Signalaufbau nicht gesetzt sein (*bSignalGrow*), wird der Meß-Timer wieder neu gestartet und die Methode mit *R_MeasureInProcess* beendet. Andernfalls wird mittels *PsdReadOut()* eine Zwischenauslesung der Meßwerte duchgeführt, die verbleibende Meßzeit neu berechnet, der Meß-Timer mit der verbleibenden Zeit gestartet und die Messung somit fortgeführt. Ist *fExposureTime* abgelaufen, ist der Meßvorgang in diesem Intervall abgeschlossen. Die Daten werden als gültig markiert und *PsdStop()* aufgerufen. Nun wird die tatsächliche Meßzeit ermittelt und die Meßdaten ein letztes Mal mittels *PsdReadOut()* ausgelesen. Jetzt werden Fehlerbedingungen geprüft: Ist die gemessene Zeit gleich 0, wird das gemeldet, gewartet, die Daten wieder als ungültig markiert und die Methode mit einem Fehlercode beendet. Gab es einen Hardware-Overflow, so wird das akustisch mitgeteilt und der Rückkehrcode auf *R_HardOverflow* gesetzt.
Zum Ende wird die Nachricht *wpJumpTo* an das Fenster gesendet.
Siehe TDevice::PollDevice()

PsdInit

Setzt das *TDevice*-Attribut *bReadyForRead* auf *FALSE* und initialisiert den PIO so, daß er mit dem PSD-Interface zusammenarbeitet. (asm, nur Win16)
Siehe TPsd::PsdInit()

PsdRead

Überprüft zuerst, ob es einen Hardware-Overflow gab (*bHardOverflow*) und ob der Detektor bereit ist, die Meßdaten auszugeben (*bReadyForRead*). Anschließend werden die Meßdaten vom PSD-Interface gelesen (asm, nur Win16) bzw. liefert lediglich *R_OK* zurück. (Win32)
TStoe_Psd::PsdReadOut()

PsdReadOut
Ermittelt dem übergebenen Parameter entsprechend den Inhalt des Meßdatenpuffers und ruft dazu *PsdRead()* auf.
TStoe_Psd::PollDevice()

PsdStart
Setzt das *TDevice*-Attribut *bReadyForRead* auf *FALSE* und startet den Meßvorgang. (asm, nur Win16)
TStoe_Psd::PollDevice()

PsdStop
Stoppt den Meßvorgang, prüft auf Hardware-Overflow und setzt *bReadyForRead* auf *TRUE*. (asm, nur Win16)
TStoe_Psd::PollDevice()

SetParameters
Liefert lediglich *R_OK* zurück.
Siehe TDevice::SetParameters()

8. Fehler, Probleme

Initialize
Weshalb wird das *TDevice*-Attribut *eHardware* nochmals auf *StoeHW* gesetzt, obwohl dies schon im Konstruktor geschieht ?

PsdStop
Zuerst wird *bReadyForRead* auf *TRUE* gesetzt und erst anschließend die Messung gestoppt. Umgekehrte Reihenfolge wäre vielleicht sinnvoller...

hReadBuf
Weshalb wird nicht das von *TPsd* geerbte Attribut *hCountBuf* verwendet ?

class TBraun_Psd

1. UML-Klassendiagramm

class TBraun_Psd	
	- bAbbruchmitShutter : BOOL
	- bDebugInfo : BOOL
<<static>>	- befehl[30][16] : BYTE
	- bRatemeter : BOOL
	- bRealLifeTime : BOOL
	+ bSetError : BOOL
	- Buffer[16] : char
<<static>>	- echo[30][16] : BYTE
	- eDataType : TPsdDataType
	- Header : struct MessDatenHeader
	- hReadBuf : HGLOBAL
	- lMeasTime : long
	- lPositionStop : long
	- nDeathTime : int
	- nDelayFast : int
	- nDelaySlow : int
	+ nErrorCode : int
	+ nHVRegelung_OK : int
	- nReadBufItems : int
	- PositionsDatenHeader[16] : long
	- uCBufferLength : UINT
	- uEnergyHigh : UINT
	- uEnergyLow : UINT
	- uEnergyScale : UINT
	- uMuxTimeDet1 : UINT
	- uPositionScale : UINT
	# BuildOperation(BYTE*, BYTE*, int) : int
	+ GetBufferSize(void) : int
	+ GetEnergyRange(UINT& ler, UINT& her) : int
	+ Initialize(void) : int
	# konvert(char) : BYTE
	# LoadHexFile(void) : int
	# Look_till_BaseAddr1(int) : int
	+ PsdInit(void) : int
	+ PsdReadOut(THowReadOutPsd) : int
	+ PsdStart(void) : int
	+ PsdStop(void) : int
	# ResetDelayTime(void) : void
	+ SetDataType(TPsdDataType) : void
	+ SetEnergyRange(UINT, UINT) : int
	+ SetParameters(void) : int
	# SynchronHexFile(BYTE&, BYTE&) : int
	+ TBraun_Psd(void)
	+ ~TBraun_Psd(void)

2. Klassenhierarchie

TBraun_Psd ist public abgeleitet von TPsd.

3. Friends

keine...

4. Files

Klassendefinition: m_psd.h
Implementation: braunpsd.cpp

5. Verantwortlichkeiten der Klasse

Kapselt den Zugriff auf die Hardware des Detektors BraunPSD.

6. Beschreibung der Attribute

bAbbruchmitShutter
Status-Flag, wird aus der ini-Datei eingelesen.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::PsdInit()*, *TBraun_Psd::PsdStop()*

bDebugInfo
Status-Flag, bestimmt ob Debuginformationen ausgegeben werden sollen, wird aus der ini-Datei eingelesen.
(Wird nicht weiter verwendet, stattdessen wird auf das von *TDevice* geerbte Attribut *bDebug* zurückgegriffen.)
TBraun_Psd::TBraun_Psd()

befehl[][]
Array, das 30 BraunPSD-Befehlscodes mit Parametern enthält.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::PsdStart()*, *TBraun_Psd::SetEnergyRange()*, *TBraun_Psd::PsdStop()*

bRatemeter
Status-Flag, wird aus der ini-Datei eingelesen. (externe Anzeige)
TBraun_Psd::TBraun_Psd()

bRealLifeTime
Status-Flag, wird aus der ini-Datei eingelesen.
TBraun_Psd::TBraun_Psd()

bSetError
Status-Flag, zeigt an, daß ein Fehler aufgetreten ist. (Wird nur geschrieben, nie ausgewertet – scheint demnach keine relevante Funktion zu besitzen.)
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::PsdStart()*, *TBraun_Psd::SetEnergyRange()*, *TBraun_Psd::PsdStop()*, *TBraun_Psd::LoadHexFile()*

Buffer[]
Temporärer Puffer in den Daten für bestimmte Attribute zwischengespeichert werden.
TBraun_Psd::TBraun_Psd()

echo[][]
Array, welches Referenzwerte für die Kommunikation mit dem BraunPSD enthält.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::SetEnergyRange()*, *TBraun_Psd::PsdStop()*

eDataType
Status-Flag, welches anzeigt, ob es sich bei den Meßwerten um Energiedaten (*PsdEnergyData*) oder Positionsdaten (*PsdPositionData*) handelt.
TBraun_Psd::SetDataType(), *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::PsdStart()*

Header
Header für das Datenfile.
TBraun_Psd::PsdReadOut()

hReadBuf
Zeiger auf den Meßdatenpuffer.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::PsdReadOut()*

lMeasTime
Zeitdauer der Messung. (Wird nie geschrieben bzw. gelesen – scheint demnach keine relevante Funktion zu besitzen.)

lPositionStop
Countstop fuer Positionskanäle. (Wird nie geschrieben bzw. gelesen – scheint demnach keine relevante Funktion zu besitzen.)

nDeathTime
Minimaler Impulsabstand, wird aus der ini-Datei eingelesen.
TBraun_Psd::TBraun_Psd()

nDelayFast
Status-Flag, wird aus der ini-Datei eingelesen.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::PsdStart()*, *TBraun_Psd::SetEnergyRange()*, *TBraun_Psd::PsdStop()*

nDelaySlow
Status-Flag, wird aus der ini-Datei eingelesen.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::PsdReadOut()*

nErrorCode
Beinhaltet Code für den aufgetretenen Fehler.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::PsdInit()*, *TBraun_Psd::PsdStart()*, *TBraun_Psd::SetEnergyRange()*, *TBraun_Psd::PsdStop()*, *TBraun_Psd::LoadHexFile()*

nHVRegelung_OK
Funktion unklar – wird nur geschrieben, nie gelesen. (Name deutet auf Hochspannungsregelung)
TBraun_Psd::PsdReadOut()

nReadBufItems
nReadBuffItems wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

PositionsDatenHeader[]
PositionsDatenHeader[] wird an keiner Stelle im Projekt benutzt, es ist auch kein Hinweis zu finden, was mit der Einführung dieses Attributes beabsichtigt war.

uCBufferLength
Größe des Meßdatenpuffers.
TBraun_Psd::GetBufferSize()

uEnergyHigh
Obere Grenze für das Energiefenster - wird aus der ini-Datei eingelesen und als Parameter für BraunPSD-Befehle verwendet.
TBraun_PSd::GetEnergyRange(), *TBraun_Psd::TBraun_Psd()*, *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::SetEnergyRange()*

uEnergyLow
Untere Grenze für das Energiefenster - wird aus der ini-Datei eingelesen und als Parameter für BraunPSD-Befehle verwendet.
TBraun_PSd::GetEnergyRange(), *TBraun_Psd::TBraun_Psd()*, *TBraun_Psd::~TBraun_Psd()*, *TBraun_Psd::SetEnergyRange()*

uEnergyScale
Wert für Energieskalierung - wird aus der ini-Datei eingelesen und als Parameter für BraunPSD-Befehle verwendet. Außerdem wird dieses Attribut in einigen Berechnungen (*uCBufferLength*, *uMaxEnergy*) genutzt.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::GetBufferSize()*, *TBraun_Psd::PsdReadOut()*, *TBraun_Psd::SetEnergyRange()*

uMuxTimeDet1
Multiplexerzeit für Detektor 1 - wird aus der ini-Datei eingelesen, die genaue Bedeutung ist unklar.
TBraun_Psd::TBraun_Psd()

uPositionScale
Wert für Positionsskalierung - wird aus der ini-Datei eingelesen.
TBraun_Psd::TBraun_Psd(), *TBraun_Psd::GetBufferSize()*, *TBraun_Psd::PsdReadOut()*

7. Beschreibung der Methoden

TBraun_Psd
Konstruktor. Lädt *asa.dll* und definiert “Schnittstellen“ zu dll-Funktionen. Anschließend werden einige Attribute mit Werten aus der ini-Datei initialisiert (*uEnergyScale*, *bAbbruchmitShutter*, *uPositionScale*, *uEnergyHigh*, *uEnergyLow*, *uMuxTimeDet1*, *bRatemeter*, *bRealLifeTime*, *nDeathTime*, *nDelayFast* , *nDelaySlow*, *bDebugInfo*) und Speicher für den Meßdatenpuffer (*hReadBuf*) allokiert.
TDList::InitializeModule()

~TBraun_Psd
Destruktor. Gibt den Speicher des Meßdatenpuffers wieder frei und schreibt einige Attribute in das ini-File zurück.
TDList::~TDList()

BuildOperation
Kommunikation mit dem Detektor - ruft Funktionen aus *asa.dll* (SetPort, GetPort) auf.
TBraun_Psd::~TBraun_Psd() TBraun_Psd::PsdReadOut() TBraun_Psd::PsdInit() TBraun_Psd::PsdStart() TBraun_Psd::SetEnergyRange() TBraun_Psd::PsdStop()

GetBufferSize
Liefert Größe des Meßdatenpuffers.
TBraun_Psd::TBraun_Psd(), *TPsd::Initialize()*

GetEnergyRange
Setzt den ersten Parameter auf *uEnergyLow* und den zweiten auf *uEnergyHigh* – ermittelt also die Größe des Energiefensters.
TPsdParameters::Dlg_OnCommand()

Initialize
Ruft *TPsd::Initialize()* auf und setzt das *eHardware* auf *BraunHW*.
TDList::InitializeModule()

konvert
Wandelt eine char-Hexadezimalziffer (‘0‘ - ‘F‘) in den entsprechenden BYTE-Wert um (0 – 15).
TBraun_Psd::LoadHexFile()

LoadHexFile
Lädt das zur Ansteuerung des Detektors von Braun mitgelieferte Programm *asa23.hex*.
TBraun_Psd::PsdInit()

Look_till_BaseAddr1
Wartet auf eine spezifische Antwort vom Detektor - ruft eine Funktion aus *asa.dll* (*GetPort*) auf.
TBraun_Psd::BuildOperation(), *TBraun_Psd::SynchronHexFile()*, *TBraun_Psd::LoadHexFile()*

PsdInit
Initialisierung der Detektorumgebung (Laden des Hex-Files zur Ansteuerung des Detektors, Konfiguration des Detektors mit Initialisierungswerten).
Siehe TPsd::PsdInit()

PsdReadOut
Dem übergebenen Parameter entsprechend wird der Inhalt des Meßdatenpuffers ermittelt.
TPsd::PollDevice()

PsdStart
Startet wahrscheinlich den Meßvorgang.
TPsd::MeasureStart()

PsdStop
Stoppt wahrscheinlich den Meßvorgang.
TPsd::MeasureStop(), *TPsd::PollDevice()*

ResetDelayTime
Genaue Funktion unklar - der Methodenname und die Implementierung lassen vermuten, das die eingestellte Verzögerungszeit im Detektor zurückgesetzt wird.
TBraun_Psd::PsdInit()

SetDataType
Setzt das Attribut *eDataType* entsprechend dem übergebenen Parameter und überprüft dabei, ob es sich um einen gültigen Typ handelt.
TBraun_Psd::TBraun_Psd(), ***TAreaScan::ShowSensorContinuous()***

SetEnergyRange
Setzt die Größe des Energiefensters im Detektor.
Siehe TPsd::SetEnergyRange()

SetParameters
Liefert lediglich *R_OK* zurück.
Siehe TDevice::SetParameters()

SynchronHexFile
Abgleich mit Detektor (Senden von Detektorbefehlen und warten auf eine entsprechende Antwort.)
TBraun_Psd::LoadHexFile()

8. Fehler, Probleme

Unterschied zwischen *hReadBuf* (*TBraun_Psd*) und *hCountBuf* (*TPsd*) ?

Buffer[]

Buffer[] sollte kein Klassenattribut, sondern eine lokale Variable sein.

SetDataType

Die Methode prüft, ob das übergebene Argument einen gültigen Wert besitzt. Da das Argument ein Enumerationstyp ist, würde der Compiler zur Übersetzungszeit schon melden, daß ein falscher Parameter verwandt wurde.

class TCommonDevParam

1. UML-Klassendiagramm

class TCommonDevParam
- Device : TDevice* - hDeviceList : HWND
+ TCommonDevParam(const int) - CanClose(void) : BOOL - Dlg_OnInit(HWND, HWND, LPARAM) : BOOL - Dlg_OnCommand(HWND, int, HWND, UINT) : void - LeaveDialog(void) : BOOL

2. Klassenhierarchie

TCommonDevParam ist public von TModalDlg abgeleitet.

3. Friends

keine...

4. Files

Klassendefinition: m_devhw.h
Implementation: counters.cpp
Ressourcen: counters.rc

5. Verantwortlichkeiten der Klasse

Dialog zur Einstellung der allgemeinen Parameter der Detektoren. (*Zähler-Konfiguration*)

6. Beschreibung der Attribute

Device

Zeiger auf den Detektor in *TDList*, für den die Parameter gesetzt werden sollen.
TCommonDevParam::TCommonDevParam(), TCommonDevParam::Dlg_OnCommand(), TCommonDevParam::CanClose(), TCommonDevParam::LeaveDialog()

hDeviceList

Handle für die Detektor-Auswahlbox des Dialogfensters.
TCommonDevParam::Dlg_OnInit(), *TCommonDevParam::Dlg_OnCommand()*

7. Beschreibung der Methoden

TCommonDevParam

Konstruktor. Setzt *Device* auf den dem übergebenen Detektor entsprechenden Eintrag in *TDList*.
TDList::SetParametersDlg*()*

CanClose

Überprüft die Daten der Controls des Dialogfensters auf bestimmte Bedingungen und ordnet diese den entsprechenden Attributen des Detektors zu. Abfragen der Checkboxen für Sound bzw. Meßfehler. Sofern nicht 0.0 eingegeben wurde, wird der eingegebene Meßfehler, die Impulsbegrenzung und die Zeitbegrenzung auf Bereichsverletzungen untersucht und gegebenenfalls auf eine der Grenzen bei entsprechender Unter- bzw. Überschreitung gesetzt

Meßfehler:
    0.001 bis 1.0
Impulsbegrenzung:
    dwLowerCountBound bis *dwUpperCountBound* (counters.cpp ⇒ 1 bis 300000)
Zeitbegrenzung:
    fLowerTimeBound bis *fUpperTimeBound* (counters.cpp ⇒ 0.1 bis 500.0 Sekunden)

TCommonDevParam::Dlg_OnCommand()

Dlg_OnInit

Initialisierung des Dialogfensters. Der Fokus wird auf das Dialogfenster gesetzt und die Auswahlbox für die Detektoren mit den Bezeichnungen der angeschlossenen Detektoren gefüllt. Abschließend werden 2 Nachrichten (*cm_ParamSet* und *cm_Initialization* (rc_def.h)) an das Dialogfenster gesendet.

Dlg_OnCommand

Allgemein: Auswerten der Nachrichten, die an das Dialogfenster gesendet werden.

- *cm_ParamSet*
  Initialisierung der Dialogelemente mit den dem aktuell eingestellten Detektor zugehörigen Werten.
  (*hDeviceList*, *id_Analysator* ???, *id_SoundSwitch*, *id_ShowCounter*, *id_Failure*, *id_StaticFailure*, *id_ExposureTime*, *id_ExposureCounts*)

- *id_ShowCounter*
  Ausführen der Methode *TCommonDevParam::CanClose()*. Sollte diese Methode mit *FALSE* zurückkehren, so wird die Nachricht *cm_ParamSet* an das Fenster gesendet. Danach wird überprüft, ob der Zähler angezeigt wird. Wenn ja, so wird an das übergeordnete Fenster die Nachricht *cm_CallExecuteCounter* gesendet. Wenn nicht, wird die Nachricht *cm_ParamSet* an das Dialogfenster gesendet.

- *cm_ActivateChanges*
  Das Detektorattribut *bSound* wird entsprechend der Dialogeinstellung (*id_SoundSwitch*) direkt gesetzt. Anschließend wird überprüft, ob der Detektor aktiv ist. Ist das der Fall, so wird die aktive Messung gestoppt, die neuen Detektorparameter gesetzt und die Messung neu gestartet.

- *id_SoundSwitch*
  Die Nachricht *cm_ActivateChanges* wird an das Dialogfenster gesendet.

- *cm_SpecificParameters*
  Setzen von devicespezifischen Parametern. Dazu wird die Detektormethode *SetSpezificParametersDlg()* aufgerufen. Abschließend wird die Nachricht *cm_ActivateChanges* an das Dialogfenster gesendet.

- *id_StaticFailure*
  Das Detektorattribut *bStaticFailure* wird entsprechend der Dialogeinstellung direkt gesetzt. Anschließend wird das Control *id_Failure* dem Detektorattribut *bStaticFailure* entsprechend eingestellt, so daß Maus- bzw. Tastatureingaben entweder verarbeitet (*TRUE*) bzw. ignoriert (*FALSE*) werden.

- *id_ChooseDevice*
  Einstellen des aktuell ausgewählten Detektors. (Ermitteln der Auswahl in der ComboBox.) Abschließend wird die Nachricht *cm_ParamSet* an das Dialogfenster gesendet

- *cm_RotateDevice*
  Es wird der in *TDList* nächste Detektor als aktueller Detektor eingestellt und die Nachricht *cm_ParamSet* an das Dialogfenster gesendet.

Alle anderen Nachrichten werden von der Basisklasse *TModalDlg* verarbeitet

LeaveDialog

Überprüft, ob der Detektor aktiv ist. Ist das der Fall, so wird die aktive Messung gestoppt, die neuen Detektorparameter gesetzt und die Messung neu gestartet. Andernfalls werden ohne Abbruch und Neustart der Messung die neuen Detektorparameter gesetzt.

8. Fehler, Probleme

Es wird auf Fensterelemente zugegriffen, die nicht in den Ressourcen definiert sind (*id_Level*, *id_Analysator*)

9. Ressourcen

CONTROL (nur solche mit einer ID)

ID	Typ	Beschriftung
id_ExposureCounts	Eingabefeld (EDIT)	*Impuls-Begrenzung* (RTEXT)
id_ExposureTime	Eingabefeld (EDIT)	*Zeit-Begrenzung* (RTEXT)
id_Failure	Eingabefeld (EDIT)	*% (1..100)* (LTEXT)
id_StaticFailure	Checkbox (BorCheck)	Meß-Fehler festlegen
id_SoundSwitch	Checkbox (BorCheck)	Sound
id_ShowCounter	Checkbox (BorCheck)	Anzeigen

PUSHBUTTON

ID	Beschriftung
cm_RotateDevice	R (nicht sichtbar ???)
cm_SpecificParameters	Weitere Einstellungen
IDOK	Ok
IDCANCEL	Abbrechen

COMBOBOX
id_ChooseDevice

class TScsParameters

1. UML-Klassendiagramm

class TScsParameters
- *Device : TRadicon
+ TScsParameters() - CanClose(void) : BOOL - Dlg_OnInit(HWND, HWND, LPARAM) : BOOL - Dlg_OnCommand(HWND, int, HWND, UINT) : void - LeaveDialog(void) : BOOL

2. Klassenhierarchie

TScsParameters ist public von TModalDlg abgeleitet.

3. Friends

keine..

4. Files

Klassendefinition: m_devhw.h
Implementation: counters.cpp
Ressourcen: counters.rc

5. Verantwortlichkeiten der Klasse

Die Klasse steuert einen Dialog zur Einstellung der Parameter des 0-dimensionalen Detektors *Radicon SCSCS* (*Settings for Radicon SCS*).

6. Beschreibung der Attribute

Device

Zeiger auf den Detektor, für den Einstellungen vorgenommen werden sollen.
TScsParameters::TScsParameters(), *TScsParameters::Dlg_OnCommand()*, *TScsParameters::CanClose()*

7. Beschreibung der Methoden

TScsParameters

Überprüft, ob ein Detektor vom Typ Radicon SCSCS an der TDList-Geräteliste aufgeführt ist, und weist ihn, falls vorhanden, **Device* zu. Falls nicht, wird eine Fehlermeldung in einer Messagebox ausgegeben.
TRadicon::SetSpezificParametersDlg()

CanClose

Überprüft die Daten der Elemente des Dialogfensters auf bestimmte Bedingungen und ordnet diese den entsprechenden Attributen des Detektors zu. Es wird kontrolliert, ob die Werte für *LowerLevel* und *UpperLevel* zwischen 0 und 1023 liegen. Ist dies nicht der Fall, so werden diese Controls auf 0 bzw. 1023 gesetzt, je nachdem, ob der Bereich über- bzw. unterschritten wurde. Des weiteren wird überprüft, ob *LowerLevel* kleiner ist als *UpperLevel*. Sollte dies nicht der Fall sein, so wird *LowerLevel* auf den Wert von *UpperLevel* gesetzt. Für *HighVoltage* wird ebenfalls untersucht, ob der gegebene Wert zwischen 0 und *MaxHighVoltage* liegt. Sollte das nicht gegeben sein, so wird *HighVoltage* auf 0 bzw. *MaxHighVoltage* gesetzt, je nachdem, ob der Bereich über- bzw. unterschritten wurde.
Wurde mindestens eine der gegebenen Bedingungen verletzt, so wird eine Nachricht (*cm_ParamSet*) an das Dialogfenster (?) geschickt und ein Signalton ausgegeben.
(Bemerkung: counters.cpp ⇒ MaxHighVoltage = 900)

Dlg_OnInit

Initialisierung des Dialogfensters. Sendet eine Nachricht (*cm_ParamSet* (rc_def.h)) an das Dialogfenster und setzt den Fokus auf dieses Fenster (Control *id_AngleRange* bekommt den Fokus).

Dlg_OnCommand

Allgemein: Auswerten der Nachrichten, die an das Dialogfenster gesendet werden.
Wird eine spezielle Nachricht (*cm_ParamSet*) empfangen, so werden die Elemente des Dialogfensters mit zugehörigen Daten gefüllt.

Control	Daten
LowerLevel (untere Schranke der Hochspannung)	wDacLowerTresh (siehe class TRadicon)
UpperLevel (obere Schranke der Hochspannung)	wDacUpperTresh (siehe class TRadicon)
HighVoltage (aktuelle Betriebsspannung)	wHighVoltage (siehe class TRadicon)

Alle anderen Nachrichten werden von der Basisklasse *TModalDlg* verarbeitet

LeaveDialog

Bedeutung ist unklar – liefert lediglich *TRUE* als Rückkehrwert.

8. Fehler, Probleme

id_AngleRange ist nicht in den Ressourcen definiert.
Bemerkung: An vielen Stellen werden Detektorattribute direkt und nicht über entsprechende Methoden gelesen bzw. gesetzt.

9. Ressourcen

CONTROL (nur solche mit einer ID)

ID	Typ	Beschriftung
id_LowerLevel	Eingabefeld (EDIT)	*Untere Schranke:* (LTEXT)
id_UpperLevel	Eingabefeld (EDIT)	*Obere Schranke:* (LTEXT)

EDITTEXT

ID	Beschriftung
id_HighVoltage	*Hochspannung* (LTEXT)

PUSHBUTTON

ID	Beschriftung
IDOK	Ok

class TPsdParameters

1. UML-Klassendiagramm

class TPsdParameters
- Device : TPsd* - OldId : int
- CanClose(void) : BOOL - Dlg_OnInit(HWND,HWND,LPARAM) : BOOL - Dlg_OnCommand(HWND,int,HWND,UINT) : void - LeaveDialog(void) : BOOL + TPsdParameters(void)

2. Klassenhierarchie

TPsdParameters ist public von TModelessDlg abgeleitet.

3. Friends

Keine.

4. Files

Klassendefinition: m_psd.h
Implemetation: m_dlg.cpp

5. Verantwortlichkeiten der Klasse

TPsdParameters implementiert einen Dialog, in dem gerätespezifische Einstellungen für einen Psd gemacht werden können. Die Klasse bietet somit die grafische Nutzerschnittstelle zu den Psd-Geräteeinstellungen. *Einstellungen für den PSD.*

6. Beschreibung der Attribute

Device

Referenz auf das Gerät, das in diesem Dialog eingestellt wird.
TPsdParameters::Dlg_OnInit(), TPsdParameters::Dlg_OnCommand(), TPsdParameters::CanClose()

OldId

Wird lediglich definiert, jedoch nie verwendet. Der Sinn ist unklar.

7. Beschreibung der Methoden

TPsdParameters

Der Konstruktor ist leer.
TAreaScan::CallLocalAction(), *TSetupAreaScan::Dlg_OnCommand()*

CanClose

Schickt zuerst eine *SetPostionRange*-Nachricht an sein Fenster. Damit werden die Werte für die Position in die entsprechenden Attribute des Gerätes geschrieben. Der Zustand der Checkbox für die Hochspannungsregelung wird entsprechend in das Geräteobjekt übertragen. Gleiches gilt für „*Links = 0“* bzw. *bReadLeftFirst*. Der Wert in „*Addiere ... Kanäle*“ wird überprüft, ob er mindestens 1 und höchstens ein Zehntel der vorhandenen Kanäle ist. Ist eine der Bedingungen nicht erfüllt, so wird der Wert entsprechend korrigiert und ein lokales Statusflag gesetzt. Der eventuell korrigierte Wert wird mittels *SetAddedChannels()* in das Geräteobjekt übertragen. Dann werden die obere und die untere Schranke für die Energieregelung ausgelesen und mittels *SetEnergyRange()* in das Geräteobjekt übertragen.
TPsdParameters::Dlg_OnCommand()

Dlg_OnInit

Überprüft zuerst, ob in der Liste der Geräte ein PsdGerät exisitiert. Wenn nicht, schickt die Methode eine *Cancel*-Message an sein Fenster, und endet mit FALSE. Sollte ein Psd vorhanden sein, wird eine Referenz auf dieses Gerät an das Attribut *Device* zugewiesen. Für die Textboxen Winkelbereich und Kanalabstand wird jegliche Eingabe deaktiviert, es ist also lediglich Anzeigen möglich. Der Focus wird auf die Textbox „*Addiere ... Kanäle*“ gesetzt. Die Methode endet mit TRUE.

Dlg_OnCommand

Wertet alle Nachrichten aus, die das Fenster empfängt. Als Reaktion auf *cm_ParamSet* liest er die aktuellen Einstellungen aus dem Geräteobjekt und setzt die anzuzeigenden Werte der Dialogelemente entsprechend. Als Reaktion auf *cm_InitializePsd* wird das Geräteobjekt veranlaßt, sich zu initialisieren. Anschließend wird ein *cm_ParamSet* an das Fenster geschickt. Als Reaktion auf *cm_ClearEnergyData* vermeldet eine Messagebox, daß dieses Feature zur Zeit nicht unterstützt wird. Als Reaktion auf *cm_SetEnergyRange* werden die aktuellen Limits für die Energiesteuerung in das Geräteobjekt übertragen. Als Reaktion auf *cm_SetPositionRange* werden die aktuellen Limits für die Position in das Geräteobjekt übertragen. Alle nicht erkannten Nachrichten werden an die Elternklasse weitergereicht.

LeaveDialog

Gibt immer TRUE zurück.

8. Fehler, Probleme

Aus welchem Grund wird diese Klasse von TModelessDlg abgeleitet und nicht von TModalDlg, wie die anderen beiden Dialogklassen ?

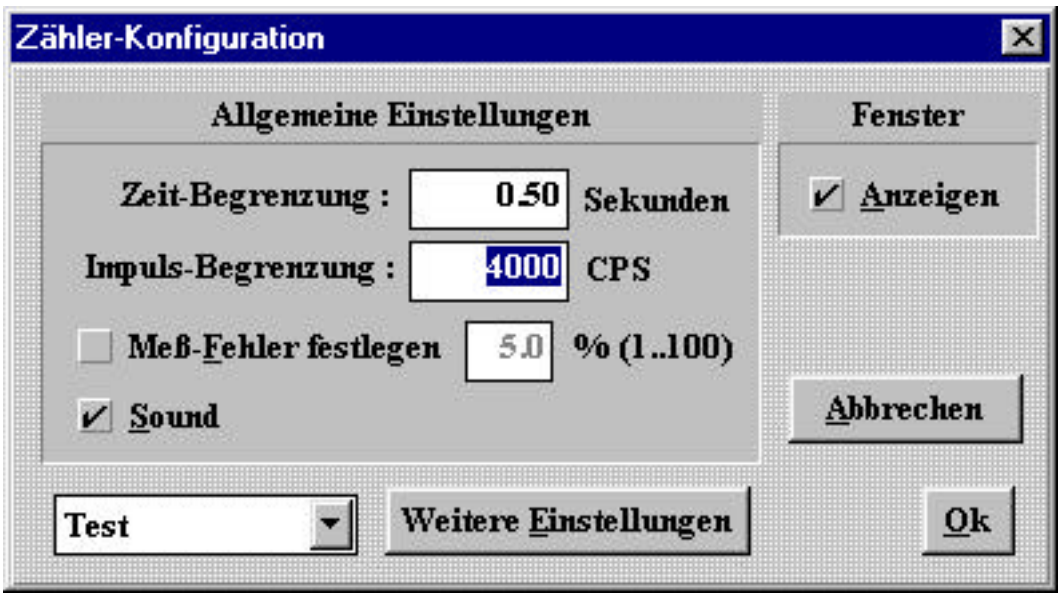
CanClose

Das lokale Statusflag *bFailure* wird im Fehlerfalle gesetzt (und auch noch mit &=), jedoch nie wieder verwendet.

Dlg_OnCommand

Die Rolle der Nachricht *cm_ClearEnergyData* läßt sich nicht erkennen. Sie wird im gesamten Programm von keiner Funktion ausgelöst.

Beschreibung des Dialogfensters „Zähler-Konfiguration“



In diesem Dialog können die Attribute eingestellt werden, die allen Detektoren gemein sind. Dies sind insbesondere:

Feld		Korrespondierendes Attribut von TDevice
Zeit-Begrenzung	id_ExposureTime	fExposureTime
Impuls-Begrenzung	id_ExposureCounts	dwExposureCounts
Textbox Meßfehler	id_Failure	fFailure
Kontrollfeld Meßfehler	id_StaticFailure	bStaticFailure
Sound	id_SoundSwitch	bSound
Anzeigen	id_ShowCounter	bDeviceOpen

Mit dem Kombinationsfeld links unten kann der Detektor ausgewählt werden, für den Einstellungen vorgenommen werden sollen. Sollten für einen Detektor weitere spezifische Einstellungen möglich sein, so kann mit der Schaltfläche „Weitere Einstellungen“ ein weiterer Dialog geöffnet werden, in dem dann die spezifischen Einstellungen vorgenommen werden können. Weiterhin kann eingestellt werden, ob für den aktuellen Detektor ein Fenster mit dem aktuellen Meßwert angezeigt werden soll (Kontrollfeld „Anzeigen“).
In der Ressourcendatei ist für diesen Dialog noch eine nicht sichtbare Schaltfläche mit der Aufschrift „R“ definiert, die offensichtlich dazu dienen sollte, jeweils zum nächsten Detektor weiterzuschalten.

Das Verhalten dieses Dialogfensters wird von der Klasse TCommonDevParam gesteuert. TCommonDevParam verarbeitet folgende Botschaften:

cm_ParamSet	Liest aus dem Detektor Characteristic bAnalysator bSound bDeviceOpen fFailure bStaticFailure fExposureTime dwExposureCounts und setzt entsprechend die zugehörigen Oberflächenelemente
id_ShowCounter	sollte TCommonDevParam::CanClose false zurückliefern, so schickt sich das Objekt selbst eine cm_ParamSet – Botschaft, andernfalls wird abhängig davon, ob bDeviceOpen gesetzt war, ein Zählerfenster geöffnet (cm_CallExecuteCounter) oder geschlossen (FORWARD_WM_MDIDESTROY). Anschließend schickt sich das Objekt eine cm_ParamSet – Botschaft.
cm_ActivateChanges	bSound wird entsprechend des zugehörigen Kontrollfeldes gesetzt Wenn Active() true zurück gibt, werden MeasureStop() SetSound() SetParameters() MeasureStart() aufgerufen.
id_SoundSwitch	Das Objekt sendet ein cm_ActivateChanges an sich selbst.
cm_SpecificParameters	Führt SetSpezificParametersDlg() aus. Anschließend sendet das Objekt ein cm_ActivateChanges an sich selbst.
id_StaticFailure	bStaticFailure wird entsprechend des zugehörigen Kontrollfeldes gesetzt Dementsprechend wird auch die id_Failure-Textbox aktiviert/deaktiviert
id_ChooseDevice	Ermittelt den Index des neu ausgewählten Detektors. Wenn dieser ungleich dem aktuellen ist, wird der neue Detektor als derjenige festgelegt, für den Einstellungen vorzunehmen sind. Anschließend schickt sich das Objekt eine cm_ParamSet – Botschaft.
cm_RotateDevice	Mit CanClose() wird geprüft, ob sich die aktuellen Einstellungen in einem konsistenten Zustand befinden. Wenn ja, wird mittels TDList::SetDevice der nächste Detektor als der aktuelle festgelegt und auch als der einzustellende Detektor festgelegt. Anschließend schickt sich das Objekt eine cm_ParamSet – Botschaft.
default	Alle anderen Botschaften werden an die Elternklasse TModalDg weitergereicht.

Fehler, Probleme

Beim Auswählen eines neuen Detektors werden die aktuellen Werte gespeichert, auch wenn dies eventuell gar nicht erwünscht war. Ein Verwerfen der Änderungen ist nicht mehr möglich.

Vor dem Speichern (und nicht gleich nach der Eingabe) wird geprüft, ob die Werte für fFailure, dwExposureCounts und fExposureTime innerhalb der Grenzen liegen.

Die Schaltfläche „Weitere Einstellungen“ ist immer aktiviert, auch wenn es gar keine weiteren Einstellungen gibt.

Wird mit dem Detektorauswahl-Kombifeld ein neuer Detektor ausgewählt, so können für diesen zwar Einstellungen vorgenommen werden; der aktuell zum Messen zu benutzende Detektor wird davon aber nicht beeinflußt. Ein eventuell aktiviertes Zählerfenster zeigt also Werte vom aktiven Detektor an, das ist möglicherweise nicht der Detektor, für den gerade Einstellungen vorgenommen werden.

Es wird versucht, entsprechend des Wertes von bAnalysator ein Control mit der ID id_Analysator zu setzen, das gar nicht existiert.

Das Control id_ShowCounter wird zweimal mit dem aktuellen Wert initialisiert, was zumindest redundant ist.

Beschreibung des Dialogfensters „Einstellungen für SCS“

In diesem Dialog können die Attribute eingestellt werden, die spezifisch für einen Radicon SCS sind.
Dies sind insbesondere:

Feld	Korrespondierendes Attribut von TRadicon
Untere Schranke	wDacLowerThresh
Obere Schranke	wDacUpperThresh
Hochspannung	wHighVoltage

Das Verhalten dieses Dialogfensters wird von der Klasse TScsParameters gesteuert. TScsParameters verarbeitet folgende Botschaften:

cm_ParamSet	Die Funktion liest die Werte aus wDacLowerThresh wDacUpperThresh wHighVoltage und füllt die entsprechenden Textfelder.
default	Alle anderen Botschaften werden an die Elternklasse TModalDlg weitergereicht

Beim Klicken auf „Ok“ werden die Werte in den Textboxen für die obere und untere Schranke geprüft, ob sie zwischen 0 und 1023 liegen. Sollte die untere Schranke größer als die obere Schranke sein, so wird sie als gleich der oberen Schranke festgelegt.

Fehler, Probleme

Es gibt in diesem Dialogfenster keine Möglichkeit, Änderungen an den Werten zu verwerfen.

Der Test, ob die untere Schranke größer als die obere ist, wird erst durchgeführt, nachdem die Werte aus den Textboxen in die korrespondierenden Attribute übernommen worden sind. Für eine gewisse Zeit können also falsche Werte in den Detektoreinstellungen stehen, mit allen Folgen für System und Hardware.

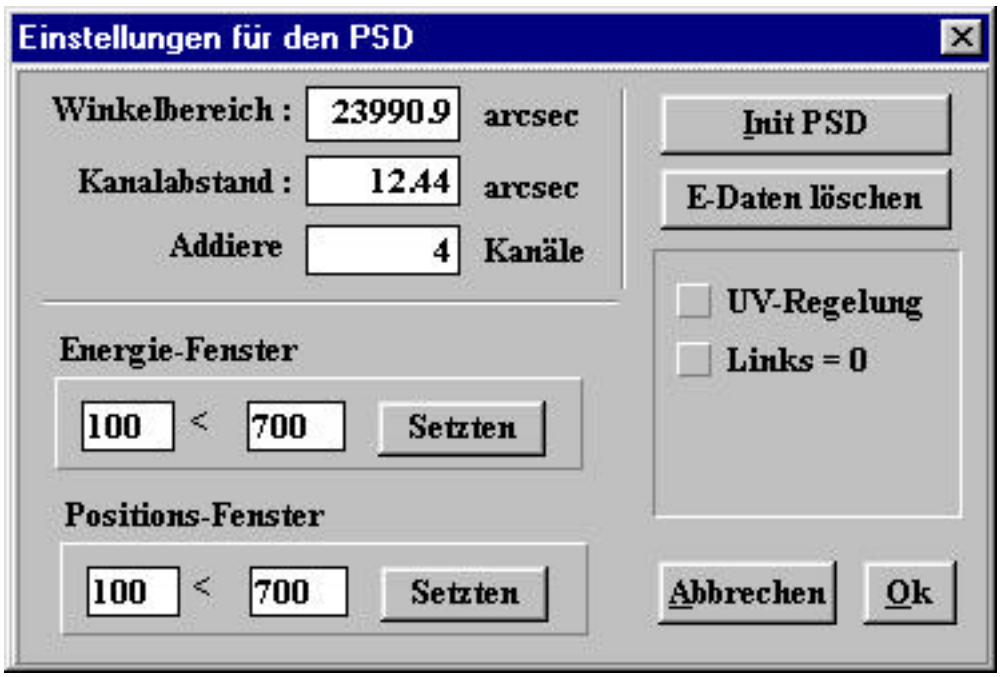
Beim Test, ob die untere Schranke größer als die obere ist, wird die Relation „>=“ verwandt. Sind beide Werte gleich, so erfolgt redundanterweise nochmals eine Zuweisung der oberen Schranke an die untere.

Die Funktion TScsParameters::Dlg_OnInit() setzt beim Öffnen des Dialogfensters den Focus auf das Dialogobjekt mit der ID id_AngleRange, das aber gar nicht existiert. Das sieht nach einem klassischen Copy-Paste-Fehler aus.

Die Werte in den Textboxen werden erst überprüft, wenn der Dialog geschlossen werden soll.

Es wird durch diese Klasse nicht sichergestellt, daß das Gerät, das hier eingestellt wird, tatsächlich ein Detektor vom Typ Radicon SCS ist.

Beschreibung des Dialogfensters „Einstellungen für den Psd“



In diesem Dialog können die Attribute eingestellt werden, die spezifisch für die eindimensionalen Detektoren vom Typ PSD sind.
Dies sind insbesondere:

Feld	ID	Korrespondierendes Attribut von TPsd
Addiere ... Kanäle	id_AddedChannels	nAddedChannels
Energiefenster (l)	id_UpperLevel	uEnergyHigh
Energiefenster (r)	id_LowerLevel	uEnergyLow
Postions-Fenster (l)	id_PositionMax	nMaxChannel
Postions-Fenster (r)	id_PositionMin	nMinChannel
UV-Regelung	id_EnergyControl	bHVRegelung
Links = 0	id_LeftConnectedZero	bReadLeftFirst

Zwei deaktivierte Felder zeigen zur Kontrolle Werte an:

Feld	ID	Angezeigter Wert
Winkelbereich	id_AngleRange	GetAngleRange()
Kanalabstand	id_StepWidth	fAngleStep

Weiterhin existieren zwei Schaltflächen mit der Aufschrift „Setzten“, deren Daseinsberechtigung zweifelhaft ist, da ihre Funktion eigentlich durch „OK/Abbrechen“ erfüllt werden sollte.

Mit dem Anklicken der Schaltfläche „Init PSD“ kann man erreichen, daß die Funktion PsdInit() aufgerufen wird.

Für die Schaltfläche „E-Daten löschen“ ist keinerlei Funktionalität implementiert, es wird lediglich eine Meldung ausgegeben, daß diese Funktion momentan nicht unterstützt wird. Die Absicht hinter diesem Control ist nicht zu erkennen.

Das Verhalten dieses Dialogfensters wird von der Klasse TScsParameters gesteuert. TScsParameters verarbeitet folgende Botschaften:

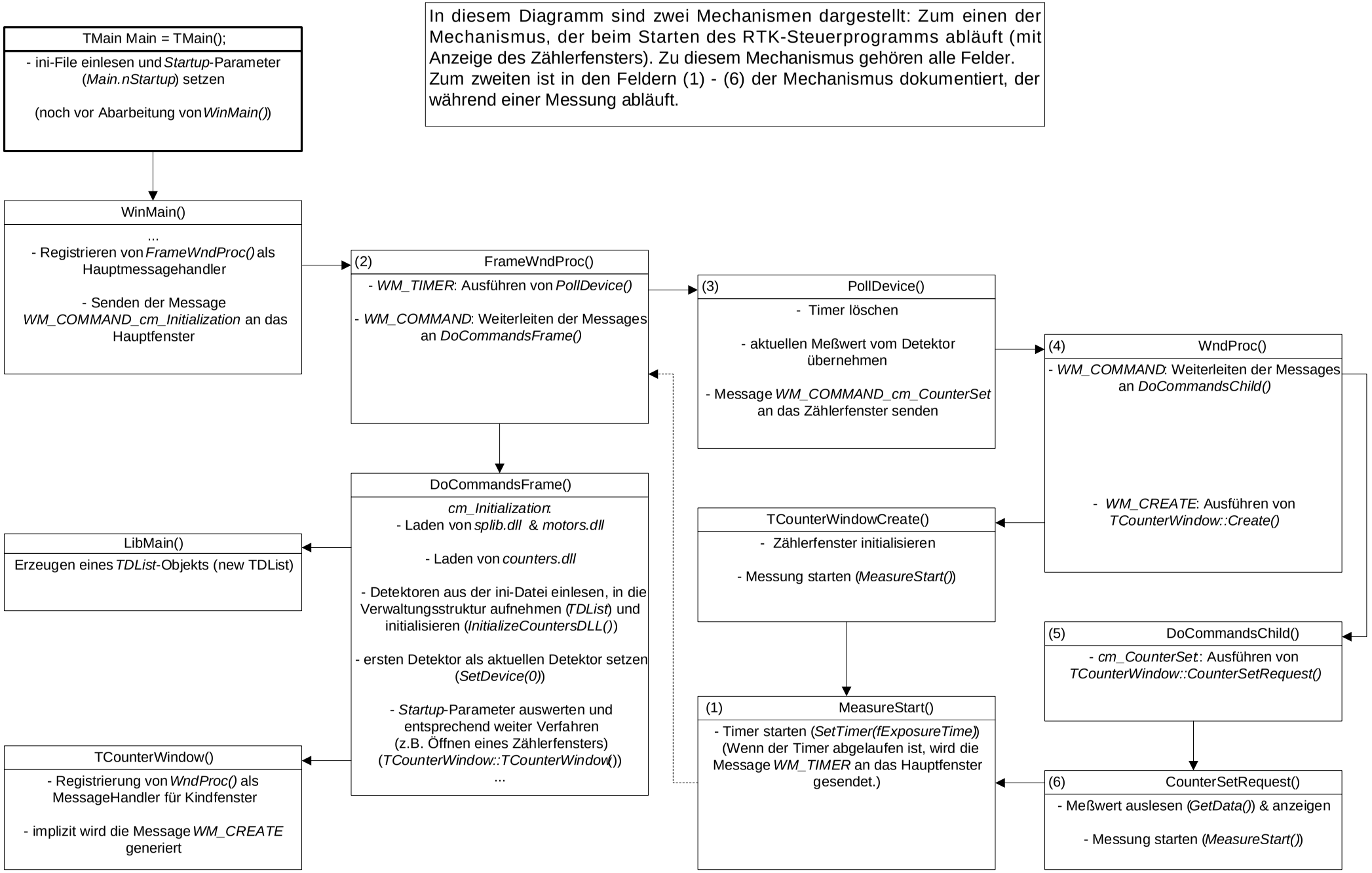
cm_ParamSet	Die Funktion liest die Werte aus GetAngleRange() nAddedChannels fAngleStep GetEnergyRange() nMaxChannel nMinChannel bHVRegelung bReadLeftFirst und füllt die entsprechenden Dialogfelder.
cm_InitializePsd	Wenn CanClose true zurückgibt, wird PsdInit() aufgerufen.
cm_ClearEnergyData	Öffnet eine MessageBox mit der schönen Ausschrift "Zur Zeit nicht unterstützt !"
cm_SetEnergyRange	Liest die Werte für das Energiefenster aus und ruft damit SetEnergyRange() auf.
cm_SetPositionRange	Liest die Werte für das Positionsfenster aus und schreibt sie in die entsprechenden Attribute.
default	Alle anderen Botschaften werden an die Elternklasse TModalDlg weitergereicht

Fehler, Probleme

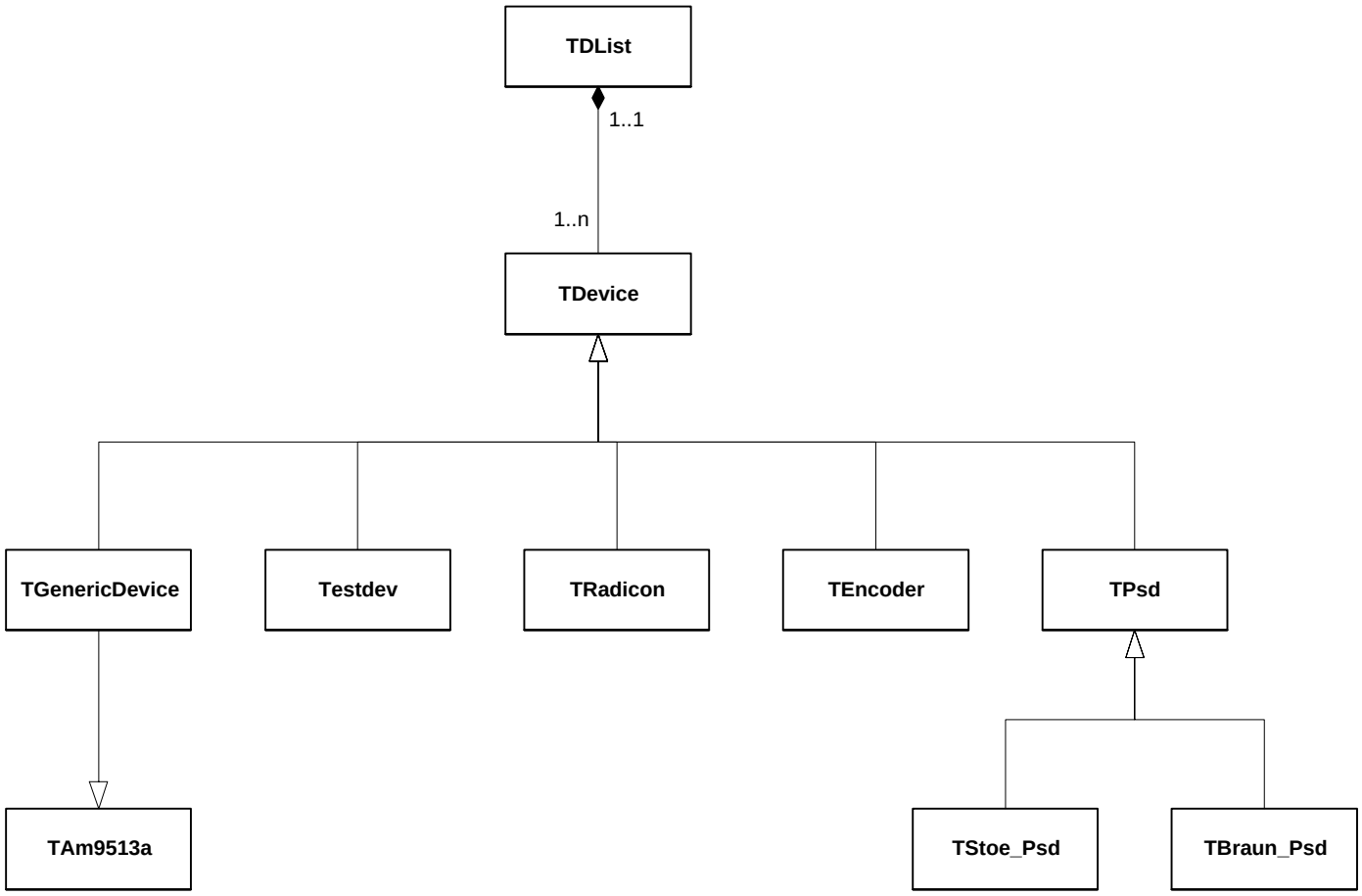
Die Werte für das Energiefenster werden ohne jede Prüfung in die Attribute geschrieben. Selbiges gilt für die Werte für das Positionsfenster.

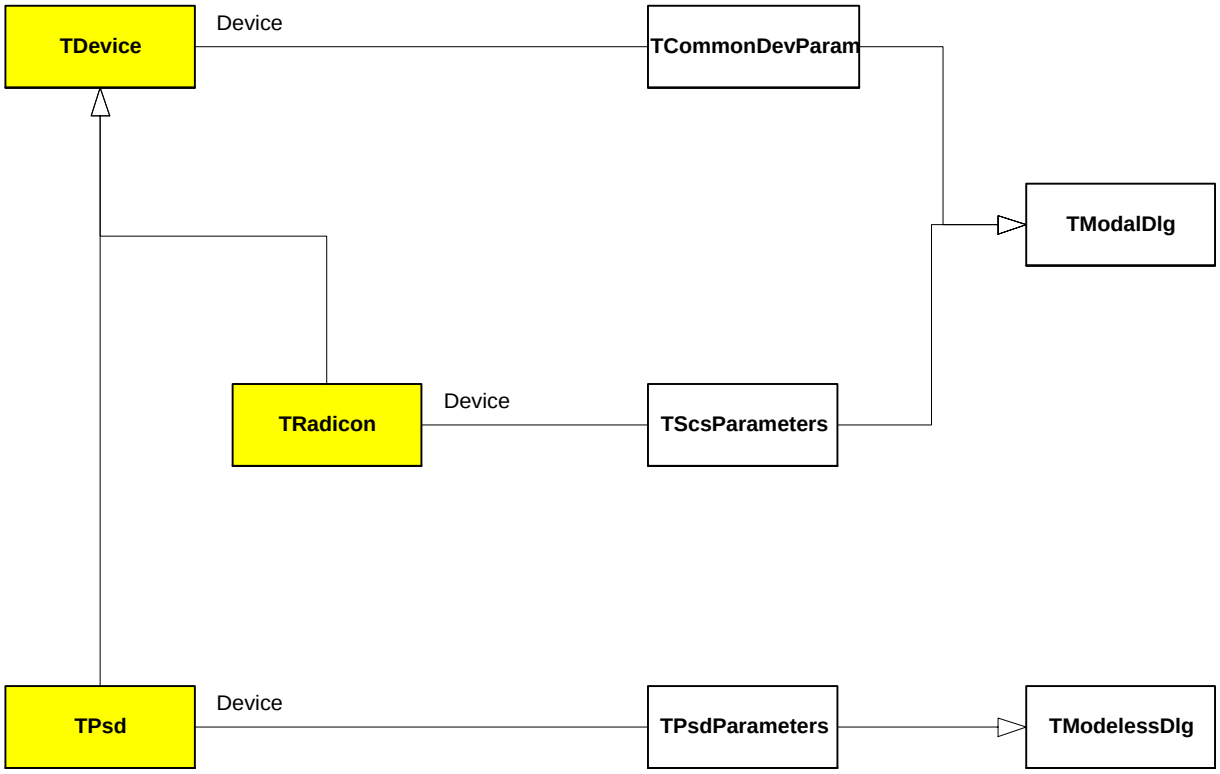
Die beiden Schaltflächen „Setzten“ sollten verschwinden und ihre Funktion von „OK/Abbrechen“ übernommen werden. Damit gäbe es auch für diese Werte die Möglichkeit, Änderungen zu verwerfen.

Die Beschriftung der Felder „UV-Regelung“ (das sollte wahrscheinlich HV-Regelung für Hochspannungs-Regelung bedeuten) und „Links = 0“ sind alles andere als selbsterklärend.


In diesem Diagramm sind zwei Mechanismen dargestellt: Zum einen der Mechanismus, der beim Starten des RTK-Steuerprogramms abläuft (mit Anzeige des Zählerfensters). Zu diesem Mechanismus gehören alle Felder.
Zum zweiten ist in den Feldern (1) - (6) der Mechanismus dokumentiert, der während einer Messung abläuft.
TMain Main = TMain();
- ini-File einlesen und Startup-Parameter (Main.nStartup) setzen
(noch vor Abarbeitung von WinMain())
WinMain()
...
- Registrieren von FrameWndProc() als Hauptmessagehandler
- Senden der Message WM_COMMAND_cm_Initialization an das Hauptfenster
(2) FrameWndProc()
- WM_TIMER: Ausführen von PollDevice()
- WM_COMMAND: Weiterleiten der Messages an DoCommandsFrame()
(3) PollDevice()
- Timer löschen
- aktuellen Meßwert vom Detektor übernehmen
- Message WM_COMMAND_cm_CounterSet an das Zählerfenster senden
(4) WndProc()
- WM_COMMAND: Weiterleiten der Messages an DoCommandsChild()
- WM_CREATE: Ausführen von TCounterWindow::Create()
DoCommandsFrame()
cm_Initialization:
- Laden von splib.dll & motors.dll
- Laden von counters.dll
- Detektoren aus der ini-Datei einlesen, in die Verwaltungsstruktur aufnehmen (TDList) und initialisieren (InitializeCountersDLL())
- ersten Detektor als aktuellen Detektor setzen (SetDevice(0))
- Startup-Parameter auswerten und entsprechend weiter Verfahren (z.B. Öffnen eines Zählerfensters) (TCounterWindow::TCounterWindow())
...
LibMain()
Erzeugen eines TDList-Objekts (new TDList)
TCounterWindow()
- Registrierung von WndProc() als MessageHandler für Kindfenster
- implizit wird die Message WM_CREATE generiert
TCounterWindowCreate()
- Zählerfenster initialisieren
- Messung starten (MeasureStart())
(1) MeasureStart()
- Timer starten (SetTimer(fExposureTime)) (Wenn der Timer abgelaufen ist, wird die Message WM_TIMER an das Hauptfenster gesendet.)
(5) DoCommandsChild()
- cm_CounterSet: Ausführen von TCounterWindow::CounterSetRequest()
(6) CounterSetRequest()
- Meßwert auslesen (GetData()) & anzeigen
- Messung starten (MeasureStart())

V. Klassenstrukturdiagramm - Detektorklassen




TDevice
Device
TCommonDevParam
TModalDlg
TRadicon
Device
TScsParameters
TPsd
Device
TPsdParameters
TModelessDlg

VI. Dokumentation und Bewertung des Interfaces

Dokumentation der C-Schnittstelle (c_layer.h)

Exportierte Funktionen

```
WINAPI   dGetExposureValues( float&, DWORD&, float& ) : BOOL
WINAPI   dlGetDevice( void ) : int
WINAPI   dlGetIdByName( TDeviceType ) : int
WINAPI   dlGetInstance( void ) : HINSTANCE
WINAPI   dlGetVersion( void ) : LPCSTR
WINAPI   dlIsDeviceValid( TDeviceType ) :  BOOL
WINAPI   dlParsingDevice( LPSTR ) : TDeviceType
WINAPI   dlSetDevice( int ) :  BOOL
WINAPI   dMeasureStart( void ) : int
WINAPI   dMeasureStop( void ) : int
WINAPI   dSetExposureValues( float, DWORD, float ) : BOOL
WINAPI   GetCounterListPtr( void ) : LPDList
WINAPI   GetIntensity_A913( void ) : float
WINAPI   GetIntensity_SCS( void ) : float
WINAPI   GetIntensity_TDC( void ) : float
WINAPI   InitializeCountersDLL( void ) : BOOL
WINAPI   InitializeTDC_Event( float, HWND ) : BOOL
CALLBACK InquireIntensity_A913( UINT, UINT, DWORD, DWORD, DWORD ) : void
CALLBACK InquireIntensity_SCS( UINT, UINT, DWORD, DWORD, DWORD ) : void
CALLBACK InquireIntensity_TDC( UINT, UINT, DWORD, DWORD, DWORD ) : void
```

Beschreibung der Funktionen

dGetExposureValues (float *Zeit, DWORD *Impulse, float *Fehlertoleranz)
Gibt die für den aktuellen Detektor eingestellten Meßlimits (Zeit, Anzahl der Impulse, Fehlertoleranz) zurück. (Nutzt *TDevice::GetExposureValues()*)
TChooseDeviceCmd::TChooseDeviceCmd()

dlGetDevice (void)
Gibt den Index des aktiven Gerätes zurück. (Nutzt *TDList::GetDevice()*)
TAreaScan::ShowSensorContinuous(), *TAreaScan::CalibratePsd()*, *TAreaScan::ExternSychronized()*, *TAreaScan::ExternSychronized()*

dlGetIdByName (TDeviceType Geräteklasse)
GetIdByName gibt den Index des Gerätes zurück, das als Parameter übergeben wird.
TAreaScanParameters::TAreaScanParameters(), *TAreaScanParameters::SetDevice()*, *TAreaScan::InitializeTask()*, *TAreaScan::CounterSetRequest()*, *TAreaScan::ShowSensorContinuous()*, *TAreaScan::CalibratePsd()*, *TAreaScan::ExternSychronized()*, *TPsdRemoteSync::TPsdRemoteSync()*, *TCalibratePsd::TCalibratePsd()*, *TAngleControl::LeaveDialog()*, *TScanParameters::TScanParameters()*, *TChooseDeviceCmd::TChooseDeviceCmd()*

dlGetInstance (void)
Gibt die Instanznummer von *Counters.dll* zurück.
m_main.cpp::ShowProgramStatus()

dlGetVersion (void)
Gibt die Versionsnummer von *Counters.dll* zurück.
m_main.cpp::ShowProgramStatus()

dllsDeviceValid (int Geräteindex)
Testet, ob der entsprechender Detektor betriebsbereit ist. (Nutzt *TDList::IsDeviceValid()*)
TAreaScanParameters::TAreaScanParameters(), *TAreaScan::TAreaScan()*,
TAreaScan::ShowSensorContinuous(), *TAreaScan::CalibratePsd()*
TAreaScan::ExternSychronized(), *TScanParameters::TScanParameters()*

dlParsingDevice (LPSTR Gerätename)
Gibt die dem übergebenen Parameter zugehörige Geräteklasse zurück bzw. *GenericDevice*.
(Nutzt *TDList::ParsingAxis()*)
TSteering::ParsingCmd()

dlSetDevice (int Geräteindex)
Legt den als Parameter übergebenen Detektor als aktiven Detektor fest.
(Nutzt *TDList::SetDevice()*)
TChooseDeviceCmd::TChooseDeviceCmd()

dMeasureStart (void)
Startet den Meßvorgang.
(Nutzt *TDevice::MeasureStart()*)
TChooseDeviceCmd::TChooseDeviceCmd()

dMeasureStop (void)
Stoppt den Meßvorgang.
(Nutzt *TDevice::MeasureStop()*)
TChooseDeviceCmd::TChooseDeviceCmd()

dSetExposureValues (float Zeit, DWORD Impulse, float Fehlertoleranz)
Setzt die Meßlimits für den aktuellen Detektor.
(Nutzt *TDevice::SetExposureValues()*)
TChooseDeviceCmd::TChooseDeviceCmd()

GetCounterListPtr (void)
Gibt Zeiger auf die Liste der verwalteten Geräte zurück. Wird benötigt um einen Zeiger auf die Detektorverwaltungsstruktur überall im Programm verfügbar zu machen.
m_main.cpp::DoCommandsFrame()

GetIntensity_A913 (void)
Gibt *Intensity_A913* zurück. Der Sinn ist unklar.

GetIntensity_SCS (void)
Gibt *Intensity_SCS* zurück. Der Sinn ist unklar.

GetIntensity_TDC (void)
Gibt *Intensity_TDC* zurück. Der Sinn ist unklar.

InitializeCountersDLL (void)
Die ini-Datei wird eingelesen, für jedes dort gefundene Gerät ein Eintrag in der Geräteliste (*TDList*) erzeugt und dieses Gerät initialisiert. (Nutzt *TDList::InitializeModule()*)
m_main.cpp::DoCommandsFrame()

InitializeTDC_Event
Der Sinn ist unklar.

InquireIntensity_A913
Setzt den Wert von *Intensity_A913*. Der Sinn ist unklar.

InquireIntensity_SCS
Liest aus der Radicon-Hardware die aktuellen Meßwerte aus und berechnet daraus die Intensität. Der genaue Sinn ist unklar.

InquireIntensity_TDC
Diese Funktion ermittelt zeitgesteuert aus den aktuellen Motorattributen einen Wert, den sie in Intensity_TDC speichert. Der Sinn ist unklar.
c_layer.cpp::InitializeTDC_Event()

Siehe Abschnitt X. 1.4. und 2.2

Gegenüberstellung der Interfaces
c_layer.h Û m_devcom.h + m_devhw.h + m_psd.h

c_layer.h

Dieses Headerfile stellt ein Interface für C-Programme zur Verfügung (Siehe Dokumentation C-Interface), indem größtenteils C++Methoden durch C-Funktionen gekapselt werden. Über den Sinn der implementierten Funktionen läßt sich streiten - es handelt sich vor allem um Funktionen zur allgemeinen Verwaltung des Moduls *counters.dll* (InitializeCountersDLL, GetInstance, GetVersion) beziehungsweise der darin implementierten Detektoren (dlGetIdByName, dlParsingDevice, dlGet/dlSetDevice, dlIsDeviceValid). Einige wenige Funktionen werden zur Konfiguration der Detektoren (dGet/dSetExposureValues) und zur Steuerung der Messung (dMeasureStart, dMeasureStop) zur Verfügung gestellt. Funktionen zur Übernahme bzw. zum Auslesen des aktuellen Meßwertes fehlen.

Als gefährlich einzustufen ist die Funktion *GetCounterListPtr()*, da hiermit ein Zeiger auf die Detektorverwaltungsstruktur geliefert wird, mit dem die C++-Schutzmechanismen leicht außer Kraft gesetzt werden können. Aufgrund des fragwürdigen Designs des Detektorsubsystems wird diese Funktion im Programm benötigt.

Abschließend sei noch bemerkt, daß es nicht nachvollziehbar ist, weshalb einige Funktionen des Steuerprogramms diese Schnittstelle nutzen, obwohl für sie die Verwaltungsstruktur *TDList* zur Verfügung steht.

m_devcom.h + m_devhw.h + m_psd.h

Die oben genannten Headerfiles stellen mit etwas gutem Willen ein objektorientiertes Interface für den Zugriff auf die Funktionalität der Detektoren zur Verfügung. Mittels dieses Interfaces besitzt man Zugriff auf folgende Klassen:
TDList, TDevice, TGenericDevice, TEncoder, TRadicon , TCommonDevParam, TScsParameters, TPsd, TStoe_Psd, TBraun_Psd , TPsdParameters.
Damit können die Funktionen sämtlicher bisher implementierter Detektoren genutzt werden. (Siehe Klassenbeschreibungen) Es stellt sich allerdings die Frage, weshalb dieses Interface genutzt werden sollte, da mit der implementierten Verwaltungsstruktur *TDList* eine viel allgemeinere Schnittstelle zur Verfügung steht.

VII. Bemerkungen zur Portabilität

Die Fragestellung der Portierbarkeit des Projektes nach Win32 aus Sicht der Detektoren ist nicht einfach nur mit „ja, möglich“ zu beantworten. Bei unseren Analysen der Quellen, die wir als zu den Detektoren gehörend erkannt haben, wurde bei bestimmten Aktionen direkt auf die Hardware zugegriffen. Im Prinzip wird jeder Zugriff auf die Hardware direkt durchgeführt und nicht über spezielle Treiber. Unter Betriebsystemen wie DOS oder WINDOWS 3.11 stellt dies auch kein Problem dar. Bei den neueren Betriebssystemen wie WINDOWS 9X und NT, bzw. 2K wird jedoch ein anderes Konzept verfolgt. Dieses besteht darin, nur noch bestimmten, privilegierten Programmen den direkten Zugriff auf die Hardware zu gestatten, damit sind vor allem Treiber und Dienste gemeint. Ein Programm wie unser vorliegendes RTK-Steuerprogramm müßte also auf einen Treiber bzw. Dienst zugreifen, um nicht durch das Betriebssystem mit einer Fehlermeldung der Form: „Software Exception - Privileged Instruction“ abrupt beendet zu werden. Doch grau ist alle Theorie, ganz so streng ist WINDOWS 9X dann doch nicht. Den Beweis dafür trat ein winziges Testprogramm (siehe unten) an, welches im wesentlichen nichts anderes macht, als mittels Inlineassembler auf einen Port zuzugreifen. Das Programm wurde unter Visual C++ 6 übersetzt und ließ uns folgendes feststellen:

1. direkte Portzugriffe sind unter WINDOWS 98 und WINDOWS 98 SE generell möglich (und somit mit ziemlicher Sicherheit auch unter WINDOWS 95)
2. direkte Portzugriffe sind unter WINDOWS NT und WINDOWS 2K nicht möglich, es kommt hier zum beschriebenen Eingriff des Betriebssystems
3. derartige Zugriffe erfordern den Inlineassembler, die momentan auch verwendeten Befehle inp() und outp() sind zumindest unter Visual C++ (WIN32) nicht mehr möglich

Fazit: Wenn man sich auf WINDOWS 98 (SE) festlegt, ist es durchaus möglich, ohne einen Treiber bzw. Dienst auszukommen, man muß jedoch die inp/outp Anweisungen durch entsprechenden Inlineassemblercode ersetzen.
Auch sei angemerkt, daß dies keinesfalls das von Microsoft favorisierte Verfahren darstellt, ein Um-/Ausbau auf das Treiberkonzept von WINDOWS ist zweifelsohne eleganter und vermutlich auch sicherer, wenngleich auch mit ungleich höherem Aufwand verbunden.

Soviel von unserer Seite zum direkten Hardwarezugriff. Ein weiteres Problem besteht unserer Meinung nach darin, daß der Typ int auf 16-bit Plattformen (also System und Compiler) eben 16-bit breit ist, auf 32-bit Plattformen 32-bit (Quelle: Borland C++ 4.5 Hilfe, Visual C++ 6 MSDN). Dieser Fakt stellt so noch kein Problem dar, jedoch dann, wenn bitweise Schiebeoperationen auf einer solchen Variablen erfolgen. Wenn man z.B. davon ausgeht, daß eine Variable 16 Bit breit ist und das 15. Bit 1 ist der Rest 0 so würde ein ShiftLeft um 1 das 15. Bit durch das 14. Bit ersetzen, der Wert der Variablen wäre nun 0. Bei einer 32 Bit breiten Variablen wäre der Wert eben nicht 0, da ja das 16. Bit den Wert des 15. aufgenommen hat. Ein solcher Fall wäre zumindest denkbar, ebenso wie der schon recht abwegige Fall, daß man sich darauf verläßt, daß der Überlauf bei 16 Bit Integer bei 65536 stattfindet. Letzteres können wir wohl ausschließen, das andere jedoch nicht mit Sicherheit. Ein Vorschlag unsererseits wäre, daß die Variablen, für die diese Vermutung zutreffen könnten, den Typ WORD bzw. int16 erhalten. Dies hätte den Vorteil, daß größere Modifikationen des Codes nicht anstünden, abgesehen von eventuellen Typumwandlungen. Diese Methode klappt zugegebenermaßen nur dann ohne größeren Aufwand, wenn es sich um einen unsigned int handelt, was jedoch meist der Fall ist.
Ebenfalls ergibt sich ein Problem von folgender Seite: Bestimmte Anweisungen und Anweisungsblöcke sind vom Entwickler durch *#ifndef WIN32 ... #endif* im Falle von WIN32 aus dem Kompilierungsvorgang herausgenommen worden, einen Elsezweig sucht man in den meisten Fällen vergebens. Dies bedeutet, daß der Entwickler von diesen Strukturen wußte oder vermutete, daß sie bei WIN32 Probleme bereiten. Des weiteren heißt das , daß

wenn kein Elsezweig vorhanden ist, an dieser Stelle nichts kompiliert wird, also bestimmte notwendige Aktionen stillschweigend wegfallen. Dessen sollte man sich bei der Portierung bewußt sein.

Hier nun das Testprogram:

```
#include <iostream.h>
void main()
{
      cout << "Programm gestartet...\n"; // Ausgabe über Streams
      _asm
      {
            mov edx,0AAh      // Wert nach edx laden
            mov eax,01h       // Wert nach eax laden
            out dx,al         // Wert aus dx (unterer Teil von edx) auf den
                              // in al gegebenen Port ausgeben
            in al,dx          // Wert von Port nach dx einlesen
      }
      cout << "Programm beendet...\n";    //fertig... hier ankommen heißt
                                          //Zugriff wurde vom OS
                                          //nicht vereitelt
}
```

VIII. Pflichtenheft + Erläuterungen zum ini - File

Pflichtenheft zum RTK-Steuerprogramm
Detektoren
Dokument:
Pflichtenheft_Detektoren.html
Dokumentversion:
1.1 (01.11.2000)
Autoren:

- U. Sacklowski (bis V1.02) unter Einbeziehung der Ergebnisse der Projektgruppe99
- Die Detektor-Gruppe
    - Jan Picard
    - Alexander Paschold
    - René Harder

Zustand:
Das Dokument ist in Bearbeitung.

---

1. Technische Hintergründe

1.1. Allgemein

Die am RTK-Meßplatz verwendeten Detektoren zählen die einfallenden Photonen. Diese Zählung kann integral oder ortsauflösend erfolgen. Im integralen Zählmodus werden alle einfallenden Photonen unabhängig von der Eintreffstelle auf dem Detektor gezählt; im ortsauflösenden Modus wird beim Zählen auch der Ort des Eintreffens berücksichtigt.
Die verwendeten Detektoren können in drei Klassen geteilt werden:

- die 0-dimensionalen Zählrohre
- die 1-dimensionalen PSD-Detektoren (**P**osition **S**ensitive **D**evice)
- die 2-dimensionalen CCD-Detektoren

1.2 Zählrohre

Zählrohre zählen die einfallenden Photonen ausschließlich integral.
Das Einsatzfeld solcher Zählrohre liegt bei:

- der Justage (manuell und automatisch), hier als Kontrollgerät hinsichtlich des Justagezustandes
- der Topographie, hier als Kontrolldetektor zwecks Überprüfung unveränderlicher Meßbedingungen
- der Diffraktometrie/Reflektometrie, hier als Meßgerät für die Ausmessung der Probe.

In allen Anwendungsfällen ist die Zählrate Photonen(Impulse) je Zeiteinheit die entscheidende Größe. Vertreter der 0-dimensionalen Detektorklasse sind der Radicon-Zähler und ein noch älterer Zähler eines ebenfalls russischen Herstellers
(siehe: Entwicklerdokumente - Pflichtenheft - Hardware).

1.3 PSDs

PSDs zählen die einfallenden Photonen ortsauflösend in einer Dimension. Dabei wird die Meßebene in Kanäle unterteilt, für die jeweils ein Meßwert ermittelt wird. Die Kanäle können zu Gruppen zusammengefaßt werden.

1.4 CCDs

CCDs zählen die einfallenden Photonen ortsauflösend in zwei Dimensionen.

2. Anwendungsfälle

2.1 Hardwareparameter einstellen

Die für den Betrieb der Detektoren notwendigen Hardwareparameter können aus einem ini-File gelesen oder über ein Dialogfeld eingestellt werden. Hierbei sind Mechanismen zu schaffen, die für die Systemintegrität kritische Hardwareparameter vor ungewollter Veränderung schützen. Benutzer, die nur mit der Messung beauftragt sind, sollen keinerlei Hardwareeinstellungen vornehmen müssen.

2.2 Meßparameter einstellen und Messung durchführen

Die für den Meßvorgang wichtigen Parameter können entweder aus dem ini-File gelesen, vom Benutzer über einen Dialog eingegeben oder über die Skriptsteuerungs-Mechanismen eingestellt werden.

3. Funktionen

3.1 Verfügbarmachung von Detektoren

Alle an einem Meßplatz angeschlossenen Detektoren werden in einem .ini-File aufgeführt. In diesem existiert für jeden Detektortyp ein eigener [Device*x*]-Abschnitt (x: 0, 1, ...) mit den typspezifischen Angaben.
Beispiel:
Wird an einem Meßplatz wahlweise der Radicon-Zähler (Type=Radicon) und das ältere russische Zählrohr (Type=Generic) benutzt, so sind für beide Zählrohre Device-Abschnitte vorzusehen. Ein dritter Abschnitt steht für ein software-simuliertes Testzählrohr. Seine Typbezeichnung muß abweichend von allen vordefinierten Typen sein, z.B. Type=Test (siehe Dokumentation der .ini-Files).
Ein Detektor ist für das Programm verfügbar, wenn er im ini-File definiert ist, technisch angeschlossen ist, und die Inititalisierung durch das Programm erfolgreich war. Sind diese Bedingungen erfüllt, steht er den anderen Subsystemen zur Verfügung und wird in den entsprechenden Dialogboxen zur Auswahl angeboten. Ausgenommen von der technischen Anschlußüberprüfung sind nur die simulierten Testdetektoren.
Zu einem Zeitpunkt kann immer nur einen Detektor genutzt werden. Beim Versuch, einen zweiten Detektor zu nutzen, erfolgt die Fehlerausschrift:

Es ist bereits ein darstellendes Fenster geöffnet. Zum Verwenden eines anderen Detektors muß die Messung mit dem aktuellen Detektor beendet werden.

3.2 Startmöglichkeiten des Zählvorganges

Der Start dieser Programmfunktion erfolgt auf zweierlei Arten:

- automatisch nach dem Programmstart, wenn die Parameter "Startup" und "Environment" im [Steuerprogramm]-Abschnitt des .ini-Files entsprechend gesetzt sind,
- über die Menüfunktion: Einstellungen - Detektoren - Detektoren...

Standardmäßig wird der Detektor aus dem Abschnitt [Device0] ausgewählt. Ist dieser nicht angeschlossen, versucht das Programm, [Device1] usw. zu nutzen. Ist kein Detektor verfügbar (auch keiner vom Typ Test), wird eine Fehlernachricht ausgegeben und es kommt zum Programmabbruch. Gleichermaßen wird verfahren, wenn kein [Device0] - Abschnitt existiert (notwendige Voraussetzung).

3.3 Auswahl des Detektors einschließlich spezieller Einstellungen

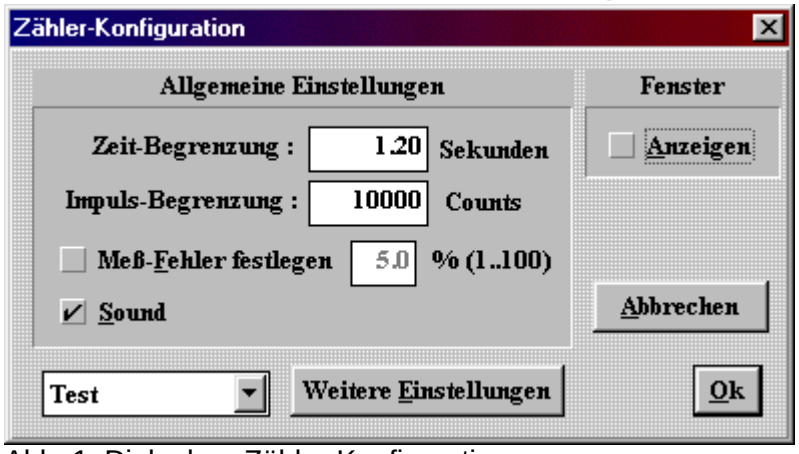


Abb. 1: Dialogbox Zähler-Konfiguration

Diese Dialogbox ist aufrufbar über:

- - PopUp-Menü im Zähler-Fenster "Einstellungen Gerät ..."
- - Haupt-Menü: Einstellungen - Detektoren - Detektoren ...

Textfelder Zeitbegrenzung und Impuls-Begrenzung:
Begrenzungsmöglichkeit für die Meßzeit und die maximale Anzahl der Impulse. Angezeigt wird fortlaufend die Anzahl der Impulse je Meßzeit-Intervall. Die maxiamle Anzahl der Impulse ist neben der Meßzeit ein zweites Abbruchkriterium. Die tatsächliche Meßzeit kann daher geringer ausfallen. Wird nämlich die maximle Impulszahl vor Ablauf der Meßzeit erreicht, so wird diese Impulszahl auf die vorgegebene Meßzeit hochgerechnet und der Zähler beginnt mit Null ein neues Meßintervall. (.ini-File: [Devicex], ExposureTime, ExposureCounts).

Kontroll- und Textfeld Meßfehler festlegen:
Festlegung des Meßfehlers in Prozent. Abschaltbar. - Bedeutung noch unklar.

Kontrollfeld Sound:
Ein- und Ausschalten der akustischen Meldung der Zählrate. Die Tonhöhe steigt mit zunehmender Zählrate. (.ini-File: [Devicex], Sound). Sollte ein Detektor kein akustisches Feedback unterstützen, so sollte auch dieses Kontrollfeld ausgegraut sein.

Kombinationsfeld Detektor:
Auswahl eines der angebotenen Detektoren.

Kontrollfeld Fenster anzeigen:
Ein- und Ausblenden des Zählerfensters.

Schaltfläche Abbrechen:
Verwerfen der Änderungen an den Werten und Schließen des Fensters.

Schaltfläche OK:
Übernahme der Änderungen an den Werten und Schließen des Fensters.

Schaltfläche Weitere Einstellungen:
Können für einen Detektor weitere spezifische Einstellungen vorgenommen werden, so kann mit diesem Button das Fenster für die spezifischen Einstellungen geöffnet werden. Gibt es für einen Detektor keine spezifischen Einstellungen zu treffen, so wird dieser Button ausgegraut angezeigt.

Die spezifischen Werte für den russ. SCS-Zähler (Type=Generic) müssen offline durch ein extra Gerät gesetzt werden.

3.3.1 Spezifische Einstellungen für einen Radicon SCS

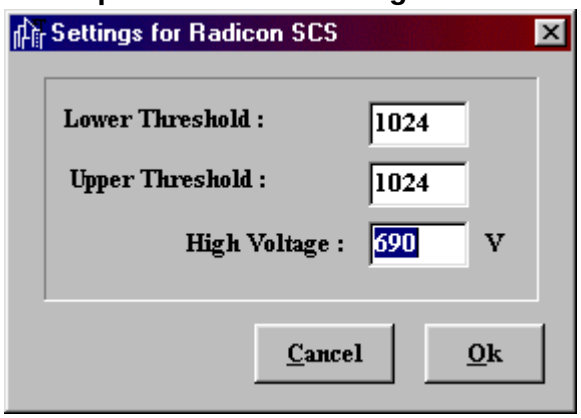


Abb. 2: Dialogbox "Settings for Radicon SCS"

Wesentliche Betriebsgrößen für diesen Detektor sind die Beschleunigerspannung und der untere und der obere Schwellwert.

Textfelder Lower Threshold, Upper Threshold:
Mit dem unteren und dem oberen Schwellwert legt man fest, in welchem Energiebereich einfallende Lichtquanten registriert werden. Dieser Bereich ist so zu legen, daß nur die durch die Röntgenbeugung entstandenen Lichtquanten gezählt werden, und damit der Ausschluß der hochenergetischen Höhenstrahlung und der niedrigenergetischen Strahlung erfolgt.
Werte für Lower Threshold, Upper Threshold werden im .ini-File im [Devicex]-Abschnitt, mit den Schlüsseln LowerThresh und UpperTresh angegeben.

Textfeld High Voltage:
Die Beschleunigerspannung liegt am Photomultiplier zwischen der Photokatode und der Anode und soll einen aus den wenigen von der Photokatode emittierten Elektronen meßbaren Strom erzeugen. Werte für High Voltage werden im .ini-File im [Devicex]-Abschnitt mit dem Schl6uuml;ssel HighVoltage angegeben.

Schaltflächen Cancel, OK:
Funktionalität wie beim Fenster "Zähler-Konfiguration".

4. Änderungswünsche, Fehler

Fehler

- Die Angaben aus dem Fenster "Settings for Radicon SCS" werden nicht in das .ini-File übernommen.

5. Offene Fragen

- Rolle des Meßfehlers (fFailure) klären
- .ini-File:
  - Werden alle Werte mit dem Schließen eines Fensters übernommen?
  - Welches ist die maximale Anzahl der Detektorabschnitte?
- Detektor:
  - Wo werden die Schwellwerte und die Arbeitsspannung für den russischen SCS abgelegt?
  - Ist feststellbar, nach welcher Zeit die Impulsbegrenzung erreicht wurde?
- Überarbeitung aus softwareergonomischer Sicht ??
  - Siehe Dokument: Auswertung der Zuarbeiten zum Dokument v1.00, - hier Angaben aus Punkt 3 und bzgl. der SW-ergon. Bemerkungen zu den einzelnen Fenstern

DEVELOP.INI
- Beschreibung der Sektion [DeviceX], zuständig für die Detektoren

Detektoren werden im ini-File als Devices geführt, ebenso im weiteren Verlauf. Die im .ini-File für ein Device festzulegenden Eigenschaften und Parameter werden mit dem Sektionsnamen (in eckige Klammern gefasst) DeviceX eingeleitet. Dabei steht X für eine ganze Zahl. Die Anzahl der maximal definierbaren Devices ist mit const int nMaxDeviceAllowed in C_Layer.cpp gegenwärtig auf 3 begrenzt, d.h. X darf maximal 2 sein. Alles was größer ist als nMaxDeviceAllowed wird stillschweigend ignoriert.
Da die einlesende Funktion GetPrivateString nicht case-sensitiv ist, ist die Groß/Kleinschreibung von Sektionsnamen und Keys irrelevant. Folgende Keys (Schlüsselworte unter einem Sektionsnamen) sind möglich (und teilweise auch erforderlich):

Type

Beschreibung	Mit diesem Key wird der Typ des Device spezifiziert, entsprechend des Typs werden unterschiedliche Parameter (Keys) benötigt und entsprechende Konstruktoren aufgerufen.
mögliche Werte	Generic, Stoe-PSD, Radicon, Braun-PSD, PSD, Encoder, Matrox, Test
Defaultwert	Generic
Detektortyp	alle
Hinweis	Der Typ Matrox ist momentan durch die Anweisung #ifdef CCD_Extension deaktiviert. Der Typ Test ist in dem Sinne nicht als Stringangabe erforderlich, vielmehr führt jeder String der nicht zu den möglichen Werten gehört (momentan also auch Matrox) zu einem Device vom Typ Test.

Name

Beschreibung	Hiermit wird dem Device ein Name zu geordnet. Dieser Name wird fortan bei der Arbeit mit dem Device angezeigt
mögliche Werte	Es können die üblichen alphanumerischen Zeichen verwendet werden, insgesamt maximal 80. Von der Verwendung von Semikola ist abzuraten, weil diese als Kommentareinleitung von GetPrivateString interpretiert werden, und nachfolgende Zeichen somit ignoriert würden.
Defaultwert	Counter
Detektortyp	alle

Sound

Beschreibung Damit wird die akustische Ausgabe der Messung ein/ausgeschaltet. Aufgrund der Quelltextanalyse gehen wir davon aus, daß ein Setzen dieses Flags einen Hardwarezugriff zur Folge hat, ein Speaker wird (de)aktiviert.

mögliche Werte 0- aus; 1- ein

Defaultwert 1 (außer bei TRadicon : 0)

Detektortyp alle

Debug

Beschreibung Mit diesem Flag wird die Ausgabe von Debugginginformationen ein- bzw. ausgeschaltet. Es wird auf die Variable *bDebug* geschrieben, diese wird im weiteren Programmverlauf an bestimmten Stellen abgefragt. Wenn sie gesetzt ist, wird die Funktion SetInfo() mit einem String ausgeführt, welcher in die Statuszeile des Steuerprogramms geschrieben wird. Dieses Flag wird nur in Verbindung mit den Detektoren verwendet.

mögliche Werte 0- aus; 1- ein

Defaultwert 0

Detektortyp alle

Hinweis Auch dieses Flag ist unserer Meinung nach für alle Detektoren (abgesehen von Testdetektoren) gedacht worden. Es wird in der Methode Initialize() eingelesen und gesetzt. Jedoch hat der Entwickler im Konstruktor TBraun_PSD dieses Vorgehen wohl vergessen und eine neue Variable samt Key eingeführt: *DebugInfo*. Auf diese wird im weiteren Verlauf jedoch nie wieder zugegriffen, statt dessen wird wie sonst üblich auch in Braun_PSD Methoden auf *bDebug* zugegriffen. (siehe Key *DebugInfo*)

ExposureTime

Beschreibung Hier wird die Länge eines Meßintervalls festgelegt. Der Wert wird in der Variablen *fExposureTime* gespeichert.

mögliche Werte 0.1- 500.0

Defaultwert 4.0

Detektortyp alle

Hinweis Für *fExposureTime* existieren die Methoden *TDevice::GetExposureValues* und *TDevice::SetExposureValues* zum Lesen und Setzen, jedoch wird mindestens dreimal aus der Klasse *TCommonDevParam* direkt auf das Attribut zugegriffen. Der Typ PSD dient unserer Meinung nach dazu einen PSD Testzähler zu erzeugen.

ExposureCounts

Beschreibung Mit diesem Key wird die maximale Anzahl der Impulse pro Meßintervall festgelegt. Der Wert wird in der Variablen *dwExposureCounts* gespeichert.

mögliche Werte 1 - 300000

Defaultwert 10000

Detektortyp alle

Hinweis Für *dwExposureCounts* existieren die Methoden *TDevice::GetExposureValues* und *TDevice::SetExposureValues* zum Lesen und Setzen, jedoch wird mindestens dreimal aus der Klasse *TCommonDevParam* direkt auf das Attribut zugegriffen

OverflowIntensity

Beschreibung Der hier angegebene Wert gibt an, welches der größte mögliche Meßwert ist.

mögliche Werte 0 - ?

Defaultwert 50000.0

Detektortyp PSD

BaseAddr

Beschreibung Hier wird die I/O-Adresse (hexadezimal) für die Kommunikation mit der Hardware angegeben. Der Wert wird im Attribut *nBaseAddr* abgelegt.

mögliche Werte 0x100 – 0x3F8 in 4h Schritten (Quelle: BraunPSD Betriebsanleitung)

Defaultwert 0x300

Detektortyp PSD

Hinweis Das Programm führt an keiner Stelle eine Überprüfung des Inhaltes von *nBaseAddr* durch.

SignalGrowUp

Beschreibung Dieses Flag gibt an, ob auch vor Messungsende Zwischendaten übernommen werden sollen.

mögliche Werte 0- nein; 1- ja

Defaultwert 1

Detektortyp PSD

HVRegelung

Beschreibung Mit diesem Flag gibt man an, ob die Hochspannung am Gerät gesteuert werden soll.

mögliche Werte 0- nein; 1- ja

Defaultwert 0

Detektortyp PSD

ReadLeftFirst

Beschreibung Mit diesem Flag legt man fest, daß beim Auslesen der Geräte von links begonnen werden soll.

mögliche Werte 0- nein; 1- ja

Defaultwert 1

Detektortyp PSD

AngleStep

Beschreibung Dieser Key gibt den Winkel an, der von einem Kanal abgedeckt werden soll.

mögliche Werte 0-?

Defaultwert 1.0

Detektortyp PSD

Unit

Beschreibung Mit diesem Key wird die Einheit der Winkelangabe festgelegt.

mögliche Werte Grad, Sekunden, Minuten, Minuts

Defaultwert Sekunden

Detektortyp PSD

AddedChannels

Beschreibung Hiermit wird die Anzahl der zu einem Meßwert zusammenzufassenden Kanäle angegeben.

mögliche Werte 1 - ?

Defaultwert 4

Detektortyp PSD

FirstChannel

Beschreibung Dieser Key gibt den ersten zu benutzenden Kanal an.

mögliche Werte 0- 4095

Defaultwert 0

Detektortyp PSD

Hinweis Wenn der Wert von *FirstChannel* größer oder gleich *LastChannel* ist, wird für *FirstChannel* =0 und für *LastChannel*=4095 gesetzt.

LastChannel

Beschreibung Hier wird der letzte zu benutzende Kanal angegeben.

mögliche Werte 0- 4095

Defaultwert 4095

Detektortyp PSD

Hinweis Wenn der Wert von *FirstChannel* größer oder gleich *LastChannel* ist wird für *FirstChannel* =0 und für *LastChannel* =4095 gesetzt.

IOAddr

Beschreibung Hier wird die I/O-Adresse für die Hardwarekommunikation mit dem Radicon-Device angegeben, sie wird in *Rdd* gespeichert.

mögliche Werte 0x100 – 0x160 in 0x10 Schritten, 0x180 – 0x1E0 in 0x10 Schritten, 0x300, 0x310, 0x330 – 0x360 in 0x10 Schritten, 0x390, 0x3E0 (Quelle: SCSCS Operator‘s Manual)

Defaultwert 0x100

Detektortyp Radicon

Hinweis Das Programm führt an keiner Stelle eine Überprüfung des Inhaltes von *Rdd* durch. Des weiteren wäre anzumerken, daß es unnötig ist, zwei Bezeichnungen für den gleichen Sachverhalt zu vergeben. Es handelt sich zwar bei IOAddr und BaseAddr nicht um den gleichen Inhalt (IOAddr ist für Radicon, BaseAddr für PSD), da sie aber nie in der gleichen Sektion zugleich vorkommen und sowieso getrennte Methoden zum Einlesen dieser Werte existieren, wäre es sinnvoll und ein weiterer Schritt in Richtung Benutzerfreundlichkeit, sich auf eine Bezeichnung zu einigen.

UpperThresh

Beschreibung Dieser Wert repräsentiert die obere Schranke für die Hochspannung, mit der das Gerät betrieben werden soll.

mögliche Werte 1 – 1023

Defaultwert 950

Detektortyp Radicon

LowerThresh

Beschreibung Dieser Wert repräsentiert die untere Schranke für die Hochspannung, mit der das Gerät betrieben werden soll.

mögliche Werte 1 – 1023

Defaultwert 150

Detektortyp Radicon

HighVoltage

Beschreibung Dieser Key gibt den Wert der Hochspannung bei Initialisierung an.

mögliche Werte 1 - 900

Defaultwert 640

Detektortyp Radicon

EnergyScale

Beschreibung Dieser Key bestimmt die Energieskalierung.

mögliche Werte 0,1,2,3

Defaultwert 2

Detektortyp Braun

AbbruchMitShutter

Beschreibung Dieses Flag setzt die Option: Messung abbrechen mit Shutter.

mögliche Werte 0- nein; 1- ja

Defaultwert 0

Detektortyp Braun

PositionScale

Beschreibung Dieser Key bestimmt die Positionsskalierung.

mögliche Werte 0,1,2,3

Defaultwert 2

Detektortyp Braun

EnergyHigh

Beschreibung Dieser Key legt die obere Grenze für das Energiefenster fest.

mögliche Werte 0(?) bis (0xFFF / (EnergyScale+1))

Defaultwert 870

Detektortyp Braun

EnergyLow

Beschreibung Hiermit wird die untere Grenze für das Energiefenster festgelegt.

mögliche Werte 0(?) bis (0xFFF / (EnergyScale+1))

Defaultwert 526

Detektortyp Braun

MuxTimeDet1

Beschreibung Dieser Key dient der Einstellung der Multiplexerzeit für den Detektor 1. Unklar ist was genau damit gemeint ist. Wir konnten diese Information lediglich aus einem einem String ziehen.

mögliche Werte 0 – 61439(0xEFFF)

Defaultwert 40

Detektortyp Braun

Ratemeter

Beschreibung Über dieses Flag wird das externe Ratemeter selektiert.

mögliche Werte 0- aus; 1- ein

Defaultwert 0

Detektortyp Braun

RealLifeTime

Beschreibung Mit diesem Flag kann zwischen Real- und Lifetimeberechnung umgeschaltet werden.

mögliche Werte 0- Realtime, 1-Lifetime

Defaultwert 0

Detektortyp Braun

DeathTime

Beschreibung Dieser Wert gibt den notwendigen Mindestimpulsabstand an.

mögliche Werte 0- 99

Defaultwert 10

Detektortyp Braun

DelayFast

Beschreibung Die Bedeutung dieses Key konnten wir nicht endgültig klären, es handelt sich jedoch vermutlich um einen Verzögerungswert, welcher bei der Kommunikation mit der Hardware einen Timeout-Wert darstellt.

mögliche Werte Ebenso fanden wir keine Anhaltspunkte für die möglichen Werte.

Defaultwert 2

Detektortyp Braun

DelaySlow

Beschreibung Die Bedeutung dieses Key konnten wir ebenfalls nicht endgültig klären, es handelt sich jedoch vermutlich auch hier um einen Verzögerungswert, welcher bei der Kommunikation mit der Hardware einen Timeout-Wert darstellt.

mögliche Werte Ebenso fanden wir keine Anhaltspunkte über die möglichen Werte.

Defaultwert 4

Detektortyp Braun

DebugInfo

Beschreibung Dieser Key soll vermutlich die Ausgabe von Debugging-Informationen aktivieren, jedoch wird er nur einmal gesetzt und im weiteren Verlauf nie wieder verwendet, was vermutlich damit zu erklären ist, daß der bereits weiter oben erläuterte Key *Debug* diese Aufgabe übernimmt. Vermutlich stellt *DebugInfo* einen Moment der Unklarheit des Entwicklers über die bereits implementierten Attribute dar, mit anderen Worten: *DebugInfo* ist überflüssig und kann bedenkenlos weggelassen werden, da auch die Routinen der PSD-Klasse mit dem Attribut *bDebug* (entsprechende Variable für den Key *Debug*) arbeiten.

mögliche Werte 0- aus; 1- ja

Defaultwert 0

Detektortyp Braun

Hinweis siehe Information weiter oben über Key *Debug*

IOAddr

Beschreibung Dieser Key dient ebenfalls der Festlegung der I/O-Kommunikationsadresse. Er wird von *TAm9513a::LoockUp* eingelesen.

mögliche Werte ?

Defaultwert 0x230

Detektortyp AM9513 (Generic)

Hinweis Siehe Hinweise oben unter BaseAddr und IOAddr

TimeCorrection

Beschreibung Die Bedeutung dieses Key konnte von uns nicht eindeutig geklärt werden. Es bleibt uns lediglich vom Namen auf die Bedeutung zu schließen, was in diesem Fall auf die Funktion einer Zeitkorrekturkonstante deutet.

mögliche Werte Ebenfalls gab es keine Anhaltspunkte für die hier möglichen Werte.

Defaultwert 1.0

Detektortyp AM9513 (Generic)

IX. Dokumentation zum Einbinden neuer Detektoren

Implementation

Die Algorithmen für den Zugriff auf den Detektor werden in einer Klasse gekapselt. Diese muß von der Klasse *TDevice* abgeleitet werden. Das Interface zwischen RTK-Steuerprogramm und dem Detektor wird im wesentlichen durch die Methoden *Initialize()*, *SetParameters()*, *MeasureStart()*, *MeasureStop(), PollDevice()* und *GetData()* gebildet. Diese Methoden werden von *TDevice* geerbt und können/sollten mit detektorspezifischem Code überladen werden.

Initialize() und *SetParameters()* dienen hauptsächlich der Initialisierung und Konfiguration des Detektors, *MeasureStart()* bzw. *MeasureStop()* dem eigentlichen Starten bzw. Beenden des Meßvorganges im Detektor, *PollDevice()* sollte den Detektor dazu veranlassen, den aktuell gemessenen Wert in einer Variablen zu speichern (*fIntensity* (von *TDevice* geerbt)), also den Meßwert zu aktualisieren und *GetData()* liefert dem Steuerprogramm genau diesen Meßwert.

Änderungen am RTK-Steuerprogramm

Am bestehenden RTK-Steuerprogramm selbst müssen folgende Veränderungen vorgenommen werden

(1) Die zu der erstellten Klasse gehörige Header-Datei muß in *Counters.cpp* eingebunden werden.

(2) Es muß dafür gesorgt werden, daß der Detektor beim Auswerten der ini-Datei erkannt wird. Dazu muß in *Counters.cpp* in der Methode *TDList::InitializeModule()* entsprechender Code eingefügt werden.

```
if ( strstr ( strupr ( buf ), "XIdentifikationsstring" ))
 {
   aDevice[ id ] = ( XKlassenname* ) new XKonstruktor(); goto EntryFound;
  }
```

Das kursiv-geschriebene ist implementationsspezifisch zu ersetzen.

(3) Um mit dem neuen Detektor auch arbeiten zu können, muß dieser in die ini-Datei aufgenommen werden. Wichtig ist dabei vor allem der Eintrag *Type=XIdentifikationsstring*.

Sollten detektorspezifische Einstellungen nötig sein, so muß zusätzlich eine Fensterklasse erzeugt bzw. eine bestehende genutzt werden, die mit Hilfe der Methode *SetSpecificParametersDialog()* aufgerufen wird.

X. Bewertung des IST-Zustandes & Gedanken zur SOLL-Architektur

1. Designfehler

1.1 Softwareschichten

Es läßt sich nur an wenigen Stellen erkennen, daß eine Aufteilung der Software in Schichten vorgesehen war. Bei TGenericDevice sieht man eine konzeptionelle Trennung von generischer Hardware und dem eigentlichen Meßvorgang. Bei TBraunPsd sieht man die Trennung von logischen und physischen Operationen auf der Hardware, bedingt durch die Tatsache, daß die physische Hardwareansteuerung in einer Bibliothek vom Hersteller gekapselt ist. Im wesentlichen sind für jeden Detektor alle Schichten in einer Klasse zusammengefaßt. Ein einheitliches Interface zwischen den Softwareschichten ist nicht zu erkennen. Das erschwert natürlich die Einbindung neuer Hardware.

1.2 Klassenstruktur

Ein Gedanke vorweg:
„ Der Schlüssel, um gute Programme zu entwickeln, besteht darin, Klassen zu entwickeln, so daß jede sauber ein einzelnes Konzept darstellt."
(Bjarne Stroustrup, Die C++-Programmiersprache, 3. Auflage, Seite 16).

Für einen Detektor sind zumindest zwei verschiedene Konzepte zu erkennen: Die Messung und die dazu erforderliche Ansteuerung der Hardware. So sollte zum Beispiel eine Messung mit einem 0-dimensionalen Zählrohr unabhängig von der speziellen Hardware sein. Einem Nutzer der Detektoren wird wahrscheinlich die tatsächlich verwendete Controllerkarte egal sein; er möchte damit messen. Im aktuellen Design der Detektoren werden diese beiden Konzepte in einer Klasse zusammengefaßt, wodurch die Detektorklassen meist sehr komplex werden. Die an den Meßplatz angeschlossenen Detektoren werden in einer Liste verwaltet. Es ist jedoch in keiner Weise dafür gesorgt, daß diese Liste einmal und nur einmal angelegt werden kann.

Ein zweiter Gedanke von Herrn Stroustrup:
„Eines der besten Werkzeuge, um Abhängigkeitsgraphen zu entwirren, ist eine saubere Trennung von Schnittstelle und Implementierung. Abstrakte Klassen ... sind dafür in C++ das Hauptwerkzeug *(Bjarne Stroustrup, Die C++-Programmiersprache, 3. Auflage, Seite 17).*
Dem steht die Tatsache gegenüber, daß es im Detektorteil keine abstrakten Klassen gibt. Konzepte wie Detektor, Messung, Psd o.ä. sollten in einer abstrakten Klasse formuliert werden, die dann von spezifischen Klassen entsprechend mit Funktionalität erfüllt werden.

Weiterhin sollten die mehrfach in verschiedener Form verwendeten Typkennungsmechanismen überarbeitet werden. Da sich der Typ im Laufe des Objektlebens nicht ändert, ist es unsinnig, ein Attribut zu führen, daß den Typ einer Klasse repräsentiert. Wenn wirklich Typkennungen vonnöten sind, sollten diese durch polymorph verwendete virtuelle Funktionen realisiert werden. In den meisten Fällen ist die Verwendung von dynamic_cast zu erwägen.

1.3 Zugriffskontrolle

In der aktuellen Version des Teilsystems Detektoren sind die Zugriffsrechte auf Attribute und Methoden der Detektorklassen zu unbedacht gesetzt. Als Folge läßt sich nicht mehr klar eingrenzen, an welcher Stelle des Programmes Attribute verändert werden. Da nun Fehler im gesamten Programm versteckt sein können, lassen sich Fehler sehr viel schwerer lokalisieren und der Wartungsaufwand steigt enorm.

Es soll nun im folgenden immer davon ausgegangen werden, daß es niemals sinnvoll ist, ein Attribut public zu definieren. Eigentlich sollten die Daten einer Klasse privat sein und nur in Ausnahmefällen, wenn abgeleitete Klassen den Zugriff benötigen, sollten Daten protected deklariert werden.
Der Hauptgrund für den jetzigen Zustand dürfte einfach in der Bequemlichkeit liegen, für jedes Attribut Zugriffsfunktionen zu schreiben.
Klarer Vorteil von Zugriffsfunktionen ist es, daß nur durch den Aufruf einer Zugriffsfunktion ein falscher Wert ein Attribut geschrieben worden sein kann und mit der Fehlersuche bei der Zugriffsfunktion begonnen werden kann. Außerdem stellt es kein Problem dar, in einer Zugriffsfunktion bei Bedarf eine einfache Debug-Meldung unterzubringen, daß das Attribut jetzt geändert werden soll und welches der neue Wert sein wird.

Deklariert man beispielsweise alle Attribute von TDevice als protected, so sieht man folgende direkte Zugriffe auf Attribute der Klasse TDevice

Funktion	**Datei**	**Attribut**	**Anzahl/Art des Zugriffes**
InquireIntensity_SCS	c_layer.cpp	fIntensity	2/ schreibend
TSetupAreaScan::Dlg_OnInit	m_arscan.cpp	Characteristic	2/ lesend
TSetupStepScan::Dlg_OnInit	m_scan.cpp	Characteristic	2/ lesend
TSetupStepScan::Dlg_OnCommand	m_scan.cpp	Characteristic	3/ lesend
TsetupContinuousScan::Dlg_OnInit	m_scan.cpp	Characteristic	1/ lesend
TsetupContinuousScan::Dlg_OnCommand	m_scan.cpp	Characteristic	1/ lesend
TMacroExecute::Dlg_OnCommand	m_steerg	bDeviceOpen	1/ lesend
TAdjustmentExecute::LeaveDialog	m_steerg	bDeviceOpen	1/ lesend
TTopographyExecute::Dlg_OnInit	m_topo.cpp	bDeviceOpen	1/ lesend
TtopographySetParam::Dlg_OnInit	m_topo.cpp	Characteristic	1/ lesend
TtopographySetParam::Dlg_OnCommand	m_topo.cpp	Characteristic	1/ lesend

Nun mag es sein, daß lesende Zugriffe auf Attribute keinen Schaden am System anrichten können. Jedoch geht nun die Flexibilität verloren, die Implementation neuen Bedürfnissen anzupassen. Außerdem ziehen Änderungen an der Implementation Änderungen in anderen Teilen des Softwaresystems nach sich.
All diesen Ärger kann man sich ersparen, wenn man den direkten Zugriff auf die Daten verbietet und den Nutzern der Daten eine ordentliche Schnittstelle präsentiert. Solange die Schnittstelle unverändert unterstützt wird, kann die Implementation an neue Bedürfnisse angepaßt werden, ohne daß auch die das Interface nutzenden Systemteile neu übersetzt werden müssen.

In diesem Falle müßten also vier neue Funktionen in das Interface aufgenommen werden:

```
class TDevice {
...
float GetIntensity()
{
  return fIntensity;
}

BOOL SetIntensity(float newIntensity)
{
  if ( Anforderungen an einen korrekten Wert für fIntensity )
  {
    fIntensity = newIntensity;
    return TRUE;
  }
  return FALSE;
}

LPSTR GetCharacteristic()
{
  return Characteristic;
}

BOOL IsOpen()
{
  return bDeviceOpen;
}
...
}
```

Klar ist allerdings auch, daß zur Bereinigung dieses Designfehlers Änderungen in Systemteilen außerhalb der Detektoren vorgenommen werden müssen, damit nämlich nur noch die neuen Interface-Funktionen verwendet werden.

(In diesen Ausführungen wurde völlig davon abgesehen, daß es nicht sinnvoll ist, die mit einem Detektor gemessene Intensität von außerhalb ändern zu lassen. Die Funktion SetIntensity sollte also besser nicht realisiert werden.)

Ähnliches kann man bei TBraun_Psd feststellen:
Hier sind die Attribute

```
int  nErrorCode;
BOOL bSetError;
int  nHVRegelung_OK;
```

public definiert. Allerdings gibt es glücklicherweisen noch keine externen Referenzen auf diese Attribute.

Bei der Untersuchung von TDevice haben wir festgestellt, daß mit friend-Beziehungen zu leichtfertig umgegangen worden ist. Im folgenden wurde untersucht, welche Zugriffe vom Compiler abgelehnt werden, wenn die betreffende Klasse nicht mehr als friend definiert ist.

friend class TAreaScan

in Methode	Verwendung von	Anzahl/Art der Verwendung
~TAreaScan	SetDisplay()	1
InitializeTask	Characteristic	1 / lesend
CounterSetRequest	bDataValid dRealTime	1 / lesend
CalibratePsd	GetDisplay() SetDisplay()	1 3
ExternSychronized	GetDisplay() SetDisplay()	1 3

friend class TDList

in Methode	Verwendung von	Anzahl/Art der Verwendung
InitializeModule	hWndFrame Characteristic	3 / lesend

friend class TCounterWindow

in Methode	Verwendung von	Anzahl/Art der Verwendung
TCounterWindow	hControlWnd	1 / schreibend
~TCounterWindow	hDisplayWnd bDeviceOpen	1 / schreibend 1 / schreibend
CanOpen	hDisplayWnd	1 / lesend
Create	bDeviceOpen hDisplayWnd	1 / schreibend 1 / schreibend
SetTitle	Characteristic	1 / lesend
CounterSetRequest	hControlWnd	1 / lesend

Nur die Deklaration von TCommonDevParam als friend von TDevice ist gerechtfertigt, da hier zum Zwecke der Trennung von Detektor und GUI zwei eng zusammenwirkende Klassen verwandt werden. Wollte man die entsprechenden Zugriffsmethoden für die Klasse TCommonDevParam freigeben, müßte man sie public deklarieren.

Um diese Probleme zu beheben, müßte man weitere Änderungen am Interface vornehmen:

1. Der konsequente Einsatz der oben beschriebenen Methode GetCharacteristic
2. Die Methoden GetDisplay() und SetDisplay() werden public definiert. Jeder Zugriff auf bDisplayWnd wird durch einen entsprechenden Ruf von GetDisplay() resp. SetDisplay() ersetzt.
3. Es wird eine Funktion GetWndFrame() als public definiert, die hWndFrame zurückgibt.

1.4 Interface

InquireIntensity_SCS
Diese Funktion ist Teil der C-Schnittstelle zu den Detektoren. Sie liest aus der Radicon-Hardware die aktuellen Meßwerte aus und berechnet daraus die Intensität. Dieser Hardwarezugriff unter Verletzung fast aller Prinzipien der Objektorientierung (Lokalität, Geheimnisprinzip, Kapselung, Interfaces) stellt einen klaren Designfehler dar.
Funktionen, mit der die Meßwerte spezifischer Detektors ausgelesen werden, sind vom Interface-Design her unflexibel und verbreitern unnötig die Schnittstelle. Für jeden neuen Detektor muß eine neue Funktion in das Interface eingefügt werden, was zusätzlich bedeutet, daß alle verwendenden Subsysteme neu kompiliert werden müssen. Werden unbedingt solche Funktionen benötigt, dann sollte auch das Prinzip der Polymorphie ausgenutzt werden.

InquireIntensity_A913
Hier gilt das unter InquireIntensity_SCS gesagte.

InquireIntensity_TDC
InitializeTDC_Event
Diese Funktionen ermitteln zeitgesteuert aus den aktuellen Motorattributen einen Wert. Es bleibt unklar, warum Motor-Funktionalitäten in der Detektoren-Schnittstelle zu finden sind. Da die beiden Funktionen im System nicht aufgerufen werden, sollten sie alsbald aus dem Interface entfernt werden.

Zusammen mit den im zweiten Teil von Abschnitt 1.3 geäußerten Gedanken müßte ein entsprechendes Interface entworfen werden, daß von außen einen vernünftigen und kontrollierten Zugriff ermöglicht, und dabei immer noch die Möglichkeit zu Veränderung der Implementierung bietet.

Im Design von TDevice erkennt man das, was Stroustrup ein „fettes Interface“ nennt. Alle angebotenen Funktionen werden von der Wurzelklasse der Detektoren realisiert, auch wenn dahinter keinerlei Funktionalität steckt. So ist es zum Beispiel möglich, bei einem Radicon-Zähler mehrere Kanäle zu einem zusammenzufassen, auch wenn das keinerlei Sinn ergibt.
Zumindest sollte ein zweites Interface geschaffen werden, daß den Zugriff auf die eindimensionalen Funktionalitäten ermöglicht.

2. Implementationsfehler

2.1 C++

Bjarne Stroustrup gibt in seinem Buch „Die C++-Programmiersprache“ unter der Überschrift „Ratschläge für C-Programmierer“ (Bjarne Stroustrup, Die C++-Programmiersprache, 3. Auflage, Seite 15) fünf Hinweise, wie man mit C++ bessere Lösungen entwickelt als mit C:

„1. Makros sind in C++ fast völlig überflüssig. Benutzen Sie const ... oder enum ..., um Konstanten zu definieren, inline ..., um den Mehraufwand von Funktionsaufrufen zu vermeiden, Templates ..., um Familien von Funktionen und Typen zu spezifizieren und Namensbereiche ..., um Namenskonflikte zu vermeiden.“
Bis auf gelegentliche Verwendung von inline-Funktionen werden diese Ratschlägen nicht berücksichtigt.

„2. Deklarieren Sie Variablen erst, wenn Sie sie benötigen, so daß Sie sie sofort initialisieren können. Eine Deklaration darf überall dort stehen, wo auch eine Anweisung stehen darf ..., sowie in for-Anweisungs-Initialisierern ... und in Bedingungen.“
Dieser Ratschlag sollte so gut wie nirgends umgesetzt worden sein.

„3. Verwenden Sie so wenig wie möglich Felder und C-Strings. Die in der C++-Standardbibliothek vorhandenen Klassen string ... und vector ... können häufig benutzt werden, um die Programmierung im Vergleich zum traditionellen C-Stil zu vereinfachen. Im allgemeinen sollten Sie nichts selber schreiben, was schon von der C++-Standardbibliothek zur Verfügung gestellt wird.“
In diesem Zusammenhang sollte man nun erwägen, die Liste aller verwalteten Geräte von einem Pointer-Array auf einen vector<TDevice*> umzustellen, und alle Stellen, an denen ein char* verwandt wird, auf string umzustellen. Das befreit insbesondere von der Verantwortung, Speicher für Strings zu reservieren und wieder korrekt freizugeben.
Außerdem erleichtert das den Einsatz von CASE – Tools, da diese i.A. mit solchen C-Konstrukten nicht umgehen können.

2.2 sonstige Implementationsfehler

Modul: *c_layer.cpp*
Die modulglobale Variable expcounts wird lediglich einmal lesend(!!!) in InquireIntensity_SCS verwendet. Mit dem darin gespeicherten Wert, der nie festgelegt worden ist, wird der neue Wert von fIntensity berechnet.

Modul: *counters.cpp*
Methode: *TDList:InitializeModule*
Der Rückgabewert von *InitializeModule()* wird nicht überprüft, d.h. wenn die Initialisierung nicht erfolgreich war, dann wird das Modul nicht geladen, *bModulLoaded* wird jedoch nicht auf FALSE zurückgesetzt, der angezeigte Status ist somit falsch!! In *M_layer.cpp* in *mlInitializeMotorsDLL()* ist der korrekte Mechanismus implementiert.

Modul: *c_layer.cpp*
Methode: *InquireIntensity_SCS*
Diese Funktion ist Teil der C-Schnittstelle zu den Detektoren. Sie zerstört einen eventuell laufenden Meßintervall-Timer, liest aus der Radicon-Hardware die aktuellen Meßwerte aus, berechnet daraus die Intensität und benachrichtigt das Zählerfenster über die Änderung.
Der neue Wert wird in das Attribut fIntensity des aktuellen Detektors geschrieben, obwohl es sich dabei möglicherweise gar nicht um einen Radicon-Detektor handelt.
Sollte eventuell eine Messung aktiv gewesen sein, so wurde sie nun für die Detektorklasse nicht nachvollziehbar abgebrochen. Das Detektorobjekt ist in einem der Realität nicht mehr entsprechenden Zustand und kann von alleine nicht mehr in einen konsistenten Zustand gelangen.

Modul: *c_layer.cpp*
Funktion: *GetIntensity_SCS*
Diese Funktion sollte vermutlich den aktuellen Meßwert des Radicon-Detektors zurückliefern. Stattdessen liefert sie den Wert der modulglobalen static-Variable Intensity_SCS zurück, der zu keinem Zeitpunkt ein Wert zugewiesen wird.
Diese Funktion wird zum jetzigen Zeitpunkt im System nicht verwendet und sollte sobald wie möglich aus dem Interface entfernt werden.

Modul: *c_layer.cpp*
Funktion: *InquireIntensity_TDC*
Funktion: *InitializeTDC_Event*
Diese Funktionen ermitteln zeitgesteuert aus den aktuellen Motorattributen einen Wert, den sie in Intensity_TDC speichern. Die beiden Funktionen werden im System nicht aufgerufen und die Variable Intensity_TDC nicht verwendet.

Modul: *c_layer.cpp*
Funktion: *InquireIntensity_A913*
Dem Namen nach könnte man vermuten, daß hier ein an die Controllerkarte Am9513 angeschlossenes Gerät ausgelesen werden sollte. Funktionalität ist aber höchstens rudimentär implementiert.

3. Externe Fehler:

Modul: m_main.cpp
Funktion: DoCommandsFrame

```
#ifndef __WIN32__
	CountersDLLInstance = GetModuleHandle("counters.dll");
#else
	CountersDLLInstance = GetModuleHandle("device32.dll");
#endif
	if(!CountersDLLInstance)
		MessageBox(GetFocus(),"Bibliothek counters.dll nicht geladen!"
                                                        ,"Fehler",MBSTOP);
	else
	{
		if(!InitializeCountersDLL())
			nStartupReport += 4;
			lpDList = GetCounterListPtr();
			lpDList->SetDevice(0);
	}
```

Problem 1:
Nach der aktuell gültigen Projektdatei wird auch unter WIN32 keine device32.dll erzeugt. Darauf ist die darauf folgende Fehlermeldung schon vorbereitet (einsprachig!)

Problem 2:
Sollte beim Initialisieren der Detektoren ein Fehler aufgetreten sein, so wird der Zeiger auf die bereits angelegte Detektoren-Auflistung mit ihrer aktuellen Adresse überschrieben (das ist zumindest redundant), und dann das Gerät 0 als das aktuelle festgelegt. Böse Falle: GetCounterListPtr() kann durchaus einen Nullpointer zurückgeben, so daß die nachfolgende Operation einen Programmabsturz mit allen Folgen hervorruft.