Zwischenbericht

DSP Labor Signalprozessor

Wintersemester 2001/2002
Institut für Elektronik und Lichttechnik

http://user.cs.tu-berlin.de/~rothm/

Gruppe: frvo
Projekt: Iris-Scan
Betreuer: Dr. Guntram Liebsch

Teilnehmer :

Halis Uzun	183069
Alexander Kordecki	181095
Fazli Ayan	178007
Hikmet Citak	180714
Hakan Uluc	192623
Andreas Hilbert	174175
Juan José Burred	215029
Marcel Roth	182917

Inhaltsverzeichnis

1 Einleitung 4

2 Iris-Scan Vorbereitung 5
2.1 Iris-Scan Prinzip 5
2.2 Iris-Scan Rechenbeispiel 6
2.3 Iris-Scan Bildaufnahmekriterien 6

3 Iris-Scan Hardware 10
3.1 Grundsätzlicher Aufbau 10
3.2 S/W-Kameramodul 10
3.3 Videomodul 11
3.4 Ansteuerung und Digitalisierung 11
3.5 Datenausgabe 12
3.6 Anmerkung 13

4 Iris-Scan Software 14
4.1 Die Steuerung der Karte 14
4.2 Übertragung und Konvertierung der Bilddaten 14
4.3 Bildskalierung 14
4.3.1 Einleitung 14
4.3.2 Ursprung 15
4.3.3 Die Skalierungsfunktion 15
4.3.4 Anmerkungen 16
4.4 Bildschärfe 17
4.4.1 Einleitung 17
4.4.2 Vorbetrachtung 17
4.4.3 Umsetzung 18
4.4.4 Diskrete Kosinustransformation 18
4.4.5 Datenerfassung 19
4.4.6 Auswertung der Transformationsdaten 22
4.4.7 Die Schärfefunktion 23
4.4.8 Anmerkung 24
4.5 Irislokalisierung 24
4.5.1 Einleitung 24
4.5.2 Ansatz 25
4.5.2.1 Einleitung 25
4.5.2.2 Anfangspunktbestimmung 26
4.5.2.3 Kontrastbilder erstellen 29
4.5.2.4 Drei-Punkte-Algorithmus 31
4.5.3 Ergebnisse 35
4.5.4 Frühere Ansätze 36
4.5.4.1 Kantenerkennung 36
4.5.4.2 Ring-Algorithmus 38
4.5.4.3 Frühere Anfangspunktbestimmungs-Ansätze 38
4.6 Bildverarbeitung 39
4.6.1 Beschneiden des Bildes 39
4.6.2 Polarkoordinaten-Transformation 40
4.6.3 Gabor-Transformation 40
4.6.4 Wavelets 41
4.6.5 Korrelation 41
4.6.6 Entscheider 42
4.7 Programmstruktur der Bildverarbeitung 42
4.7.1 Einleitung 42
4.7.2 Hauptprogramm 43
4.7.3 Speicherbereiche 43

4.7.4 Ablauf 43
4.7.5 Iris-Code-Gewinnung 44
4.7.5.1 Speicherbereiche 44
4.7.5.2 Ablauf 45
4.7.6 Anmerkungen 45

5 PC-Schnittstelle 47
5.1 Einleitung 47
5.2 RTDX 47
5.2.1 Einführung in RTDX 47
5.2.2 RTDX Target Library API 48
5.2.3 RTDX COM API 49
5.2.4 Übertragung : Target -> Host 49
5.2.5 Übertragung : Host -> Target 49
5.3 Targetapplikation 50
5.3.1 Übersicht 50
5.3.2 Funktionsdefinitionen und Beschreibungen 52
5.3.3 Programmablauf 54
5.3.4 Probleme 55
5.4 Hostapplikation 55
5.4.1 Übersicht 55
5.4.2 Beschreibung der einzelnen Pfade 57
5.4.3 Benutzte Funktionen 57
5.4.4 Vergleiche 57

6 Anhang Quellcodes 59
6.1 Bildschärfe 59
6.2 Scaling 61
6.3 Irislokalisierung 62
6.4 Programmstruktur 73
6.4.1 Hauptprogramm 73
6.4.2 Code-Gewinnung 76
6.4.3 Linker Command File 78
6.5 Targetapplikation 79
6.6 Hostapplikation 82

7 Referenzen 91

1. Einleitung

Iris-Scan ist eine biometrische Überprüfung eines persönlichen Identifikationsmerkmals. Hier handelt es sich um das Iris des Menschen, der für die Authentifizierung zuständig ist.
Verglichen mit anderen Methoden der Authentifizierung, wie z.B. Fingerabdruck, Chip/Magnet-Karte u.s.w. besitzt die Iris-Scan Methode eine überragend verfälschungssichere Möglichkeit. Auf der Welt gibt es höchstwahrscheinlich keinen 2. Iris, die sich in mathematischen Details ähnlich zueinander stehen. Dadurch ergibt sich für diese Methode eine sehr hohe Sicherheit gegen Manipulation und Verfälschung des persönlichen Identifikationsmerkmals.

Im diesjährigen DSP-Labor haben wir uns die Aufgabe gestellt, ein Iris-Scan Prototyp auf der Basis des DSP-Prozessors zu entwickeln. Das Projekt beinhaltet folgende Schwerpunkte :

1. **Iris-Scan Vorbereitung**
   (Bildaufnahmekriterien)
2. **Iris-Scan Hardware**
   (Unterstützte und verwendete Hardware)
3. **Iris-Scan Software**
   (Der eigentliche Iris-Scan Programmablauf)
4. **PC-Schnittstelle**
   (Kommunikation zwischen DSP und PC)

2. Iris-Scan Vorbereitung

Die Iris-Scan Bildverarbeitung ist im Grunde ein komplexes Thema. Viele Zwischenschritte sind nötig um das gewünschte Resultat zu erreichen. In diesem Abschnitt wird kurz und oberflächlich das Prinzip erklärt, wie in unserem Projekt der Iris-Scan arbeitet und später dementsprechend auch im DSP funktioniert.

2.1 Iris-Scan Prinzip

Ausgehend von einer S/W Kamera aufgenommenes Bild wird die Bildverarbeitung durchgeführt. In Bild 2.1.1 erkennt man eine unerwünschte Lichtreflexion an der Regenbogenhaut.

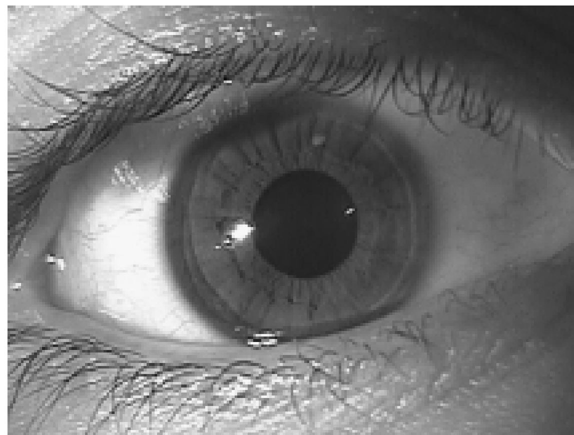


Bild 2.1.1 Originalbildaufnahme mit einer Lichtreflexion

In Bild 2.1.2 wird der wichtigste Bereich, hier die Regenbogenhaut, aus dem Bild herausgeschnitten. Die Positionierung und Erkennung des Iris liefert die Funktion Irislokalisierung.

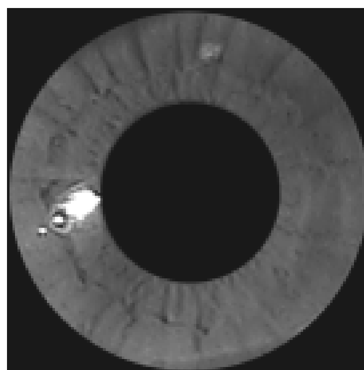


Bild 2.1.2 Regenbogenhaut Ausschnitt

Das Bild wird in Polarkoordinaten umgewandelt, so dass der Regenbogenhaut-Ring in eine Rechteckform transformiert wird.

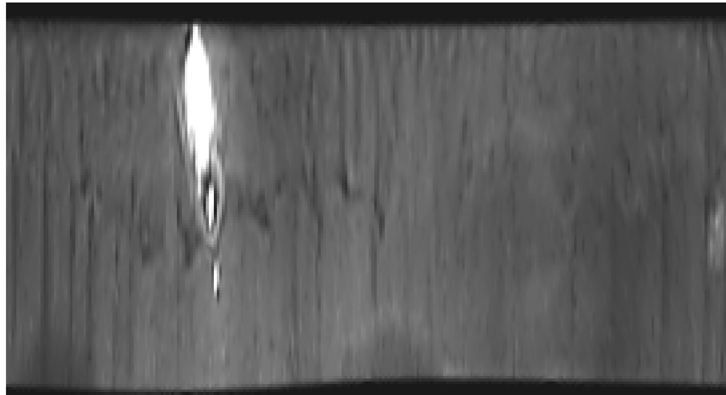


Bild 2.1.3 Polar-Transformation

Als letzter Abschnitt wird der Polar-Transformierte Iris auf eine kompakte Form gebracht und somit komprimiert, wie in Bild 2.1.4 . Die Komprimierung erfolgt mittels Gabortransformation. Der Iriscode ist somit erzeugt worden und besteht aus einigen Bytes (256 Bytes mit Gabor-Transformation).

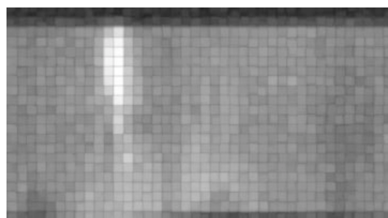


Bild 2.1.4 Iriscode

Wie man hier sieht, verdeckt die Lichtreflexion ein Teil des Iris-Merkmals. Diese kann natürlich den Iriscode verfälschen. Wie man dieses Problem umgeht, wird später erläutert.

2.2 Iris-Scan Rechenbeispiel

In das Polarform umgewandelte Bild besitzt eine Auflösung von 512 x 128 Pixel mit 8 Graustufen. So beträgt die Bildinformation für ein Iriscode 524288 Bit.
Auf der Welt leben ca. 10^{10} Menschen. Der obige Iriscode würde somit eine maximale Kombination für

$$8^{512 \times 128} \cong 8 \cdot 10^{59184}$$

Iris Merkmale zulassen, also mehr als gigantisch genug, um die Erdbevölkerung zu integrieren. (Ohne Gabor-Transformation!)

2.3 Iris-Scan Bildaufnahmekriterien

Eines der wichtigsten Gesichtspunkte der Iris-Scan Methode ist die Aufnahme bzw. die Aufzeichnung eines Irisbildes mit der Kamera. Eine unsaubere Aufzeichnung des

Iris-Bildes führt zu keinem erwünschten Ergebnis des Iris-Scans. Jede Bemühung wird fehlschlagen um daraus ein zufriedenstellendes Identifikationsmerkmal zu erzielen.
Um ein zufriedenstellendes Identifikationsmerkmal zu erreichen, müssen folgende Kriterien aufgestellt und eingehalten werden :

1. Lichtreflexion

Umgebungslicht, wie z.B. Fenster, dürfen nicht das Auge Blenden. Sonst entstehen Lichtreflexionen am Auge, die später für die Codierung eine Verfälschung des Iris-Identifikationsmerkmal mit sich bringt.

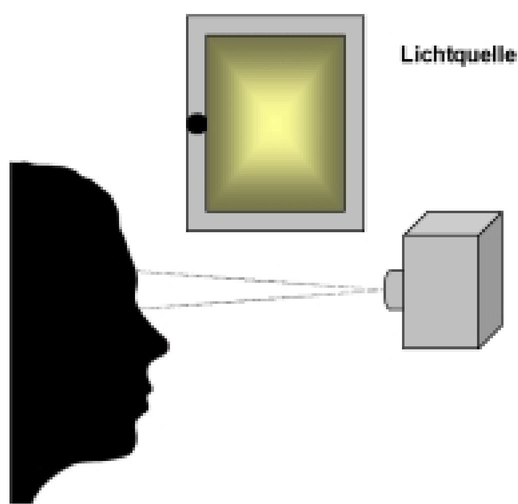


Bild 2.3.1 Iris-Scan Bildaufnahme mit Lichtquelle

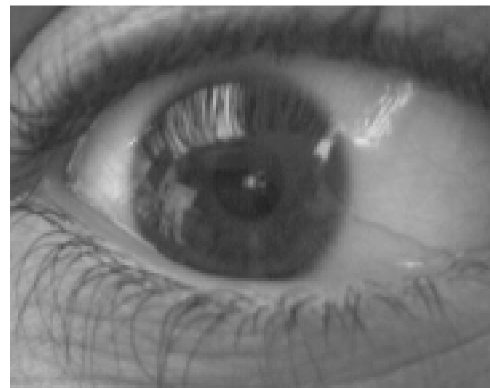


Bild 2.3.2 Lichtreflexion am Auge

2. Abdeckung

Um die unerwünschte Lichtreflexionen zu eliminieren, wird eine Lichtabdeckung aus schwarze Pappe angebracht. Somit entstehen keine Lichtreflexionen mehr am Auge. Leider geht somit auch die Helligkeit verloren, womit die Aufnahmen unbrauchbar bleiben.

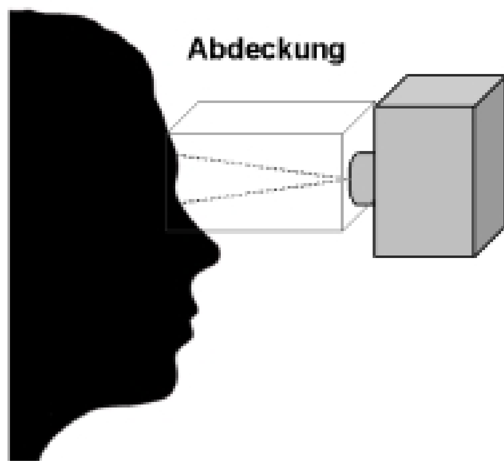


Bild 2.3.3 Iris-Scan Bildaufnahme mit Abdeckung

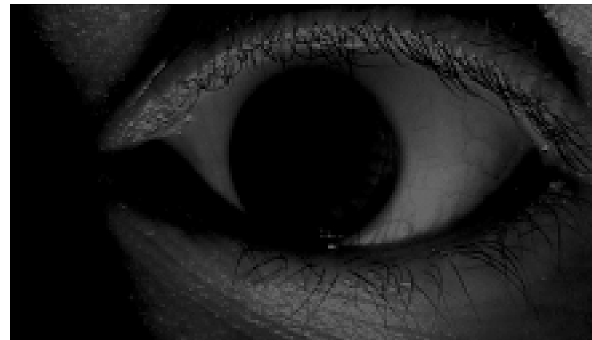


Bild 2.3.4 Bild zu dunkel

3. Bildhelligkeit

Ziel ist es nun, eine Beleuchtung anzubringen, die das Auge für die Kamera entsprechend gut ausleuchtet ohne das Auge zu Blenden.
Ein Infrarot-Scheinwerfer erfüllt diese Aufgabe souverän. Sie besteht aus einzelnen IR-LEDs, die an eine externe Spannungsquelle angeschlossen werden. Da die Kamera schon auf IR-Licht sensibel ist, können somit die Augenpartien bei völliger Dunkelheit sehr gut ausleuchten, ohne das Auge zu Blenden. Die Irismerkmale können besonders gut erkannt werden.

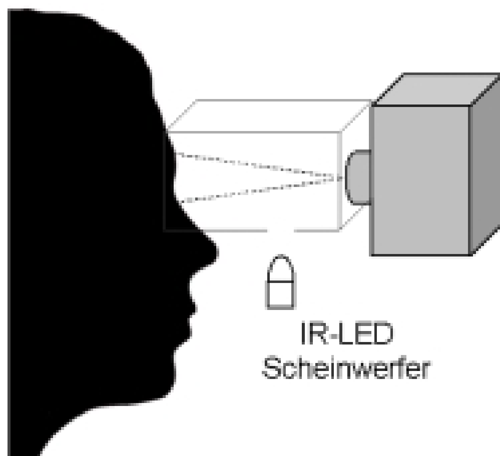


Bild 2.3.5 Iris-Scan Bildaufnahme mit IR-Scheinwerfer

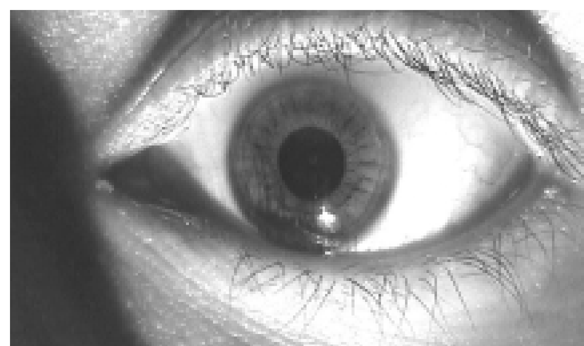


Bild 2.3.6 Blendenfreie Ausleuchtung mit Punktreflexion

Unvermeidbar ist bei dieser Anordnung die punktartige Lichtreflexionen. Sie entstehen auch bei einem IR-Licht. Hier gibt es die Möglichkeit, die IR-LED so zu positionieren, dass später der reflektierende Anteil abgeschnitten werden kann.

4. Bildschärfe

Nachdem der Aufbau für die Bilderfassung festgelegt worden ist entsteht ein sekundäres Problem : die Bildschärfe.
Die Kamera ist mit seinem optional eingebauten Makro Objektiv in der Lage, ca. 1mm Schärfenbreite bei einem Abstand von 8cm zu erzeugen. D.h. , bewegt sich die Person sein Auge um 1 mm nach hinten oder nach vorne, wird das Bild völlig unscharf. Somit ist die Aufnahme auch unbrauchbar, weil die Irismerkmale verschwimmen.

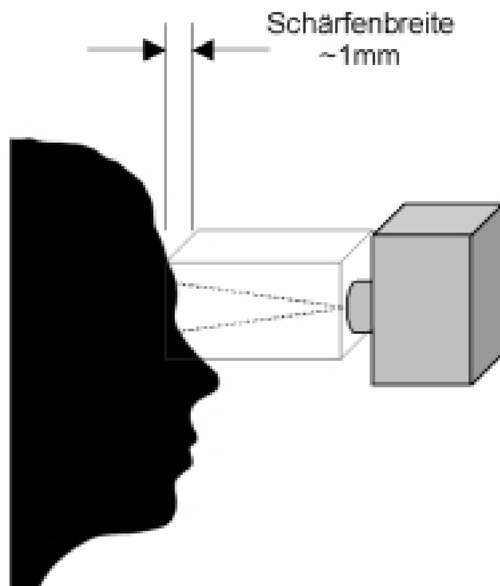


Bild 2.3.7 Iris-Scan Bildaufnahme mit 1mm Schärfenbreite

Bild 2.3.8 Verschwommene Irismerkmale

3. Iris-Scan Hardware

3.1 Grundsätzlicher Aufbau

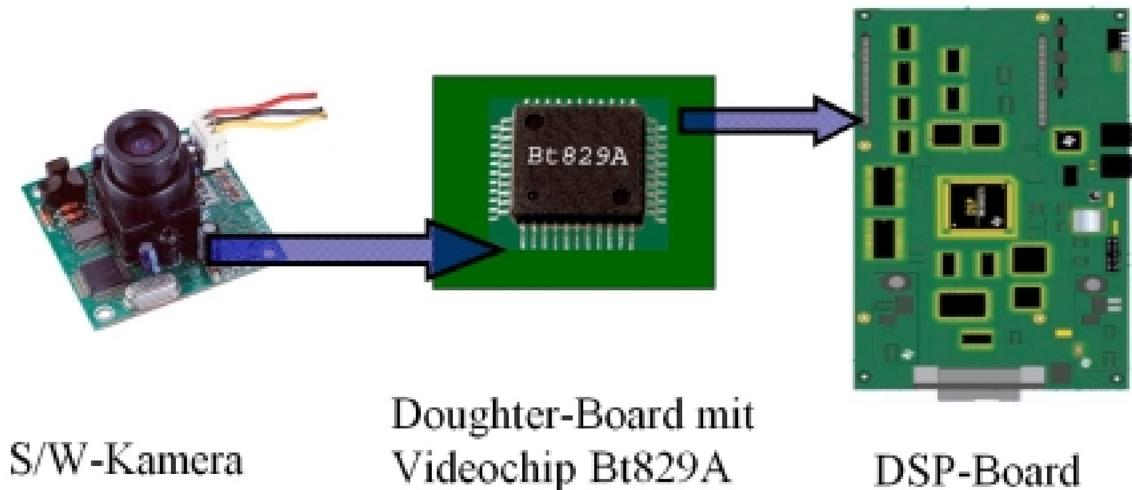


Bild 3.1 Iris-Scan Aufbau

Die S/W-Kamera erfasst, unter den genannten Kriterien, das Auge der Person. Das Bild wird über den Analogen-Videoausgang der Kamera ausgegeben. Der Videochip am Videomodul wandelt die analoge Bildinformation in digitale Signale um. Da das Videomodul auf der DSP-Karte steckt, werden die gewandelten Daten dementsprechend mit dem DSP-Prozessor verarbeitet und auf einem PC ausgegeben. Die DSP-Karte und der PC sind über die Parallel-Schnittstelle miteinander verbunden.

3.2 S/W-Kameramodul

Im Laufe des Projekts entschieden wir uns für ein kompaktes Schwarz/Weiss Kameramodul (aus dem Hause Conrad). Im Vergleich zu dem DV-Camcorder besitzt die S/W-Kamera folgende Merkmale, die letztendlich für uns ausschlaggebend sind :

1. Optionales Makro-Objektiv
2. Empfindlich gegen IR-Licht
3. Kompakte Bauweise

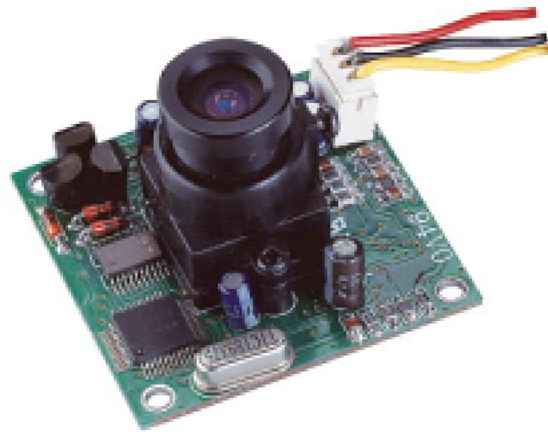


Bild 3.2 S/W Kameramodul

Technische Daten: Spannungsversorgung 12 V = (9 - 15 V) · Stromaufnahme ca. 100 mA · Ausgangspegel 1 V_{pp} / 75 Ω · Zeilenfrequenz 15,625 kHz · Bildfrequenz 50 Hz · Horizontalauflösung 380 Zeilen · Auflösung 291000 Pixels (500 [H] x 582 [V] · Lichtempfindlichkeit 1 Lux · Objektiv: f = 3,6 mm (Brennweite) / F = 2,0 mm (Blende) · Bildwinkel (diagonal) ca. 92 °C · Automatische Blendeneinstellung · Arb.-Temperaturbereich -10 °C bis +45 °C · Gewicht ca. 20 g · Abm.: 40 x 40 x 27 mm.

3.3 Videomodul

Das von uns verwendete Videomodul basiert auf dem Videochip Bt829A.
Dieser Chip wandelt ein analoges Videosignal in einen digitalen Pixelstrom, der dem DSP in Form eines korrekt beschriebenen Datenpuffers zur Weiterverarbeitung zur Verfügung gestellt werden soll.

3.4 Ansteuerung und Digitalisierung

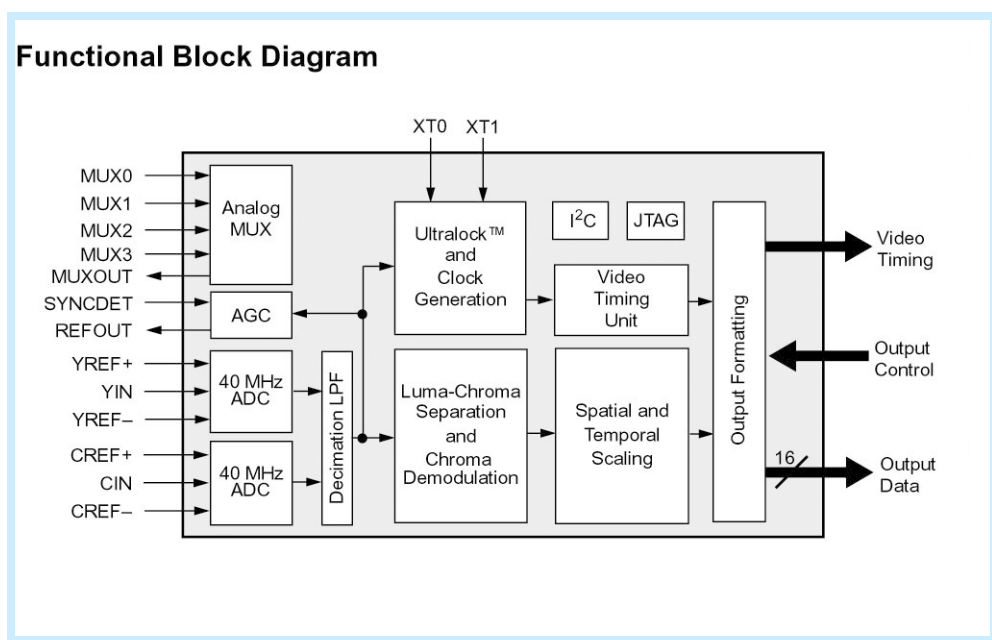


Bild 3.4 Vereinfachtes Blockdiagramm des Bt829A

Die Steuerung des Videochips erfolgt über ein i²C-Interface, womit die internen Register abgefragt (***bt_get_reg()***) und gesetzt (***bt_set_reg()***) werden können.

Das analoge Videosignal liefert uns ein Schwarz/Weiß-Kameramodul.

Aus dem anliegenden Oszillatorsignal mit einer Frequenz von 35,47MHz (= 8*PAL fsc) generiert der Videochip 2 Clocks (CLKx1 und CLKx2), deren Raten der Oszillatorfrequenz bzw. ihrer Hälfte entsprechen. Die A/D-Wandlung des Composite-Videosignals erfolgt direkt mittels Flash A/D-Wandlern bei CLKx2 also mit mehr als der doppelten zu erwartenden Videobandbreite.

Für die Verwendung von PAL-Einganssignalen an CIN wird das Input-Format-Register IFORM mit den Default-Werten gesetzt (Auto-Formaterkennung).

Die Decodierung des Videosignals wird nach der Digitalisierung durchgeführt. Zunächst erfolgt eine Tiefpassfilterung und eine anschließende Dezimierung der Rate auf CLKx1. Dann werden Chrominanz- und Luminanzsignal separiert (NotchFilter & Bandpass). Aus dem Chrominanzsignal werden per QAM-Demodulation die Farbdifferenzsignale U und V gewonnen.

Anschließend werden Einstellungen verschiedener Video-Filter für Kontrast, Helligkeit etc. umgesetzt. Die entsprechenden Register 0x0A bis 0x0F werden bei uns mit Standardwerten gesetzt.

Danach erfolgt die örtliche und zeitliche Skalierung des Videodatenstroms. Dazu müssen die entsprechenden Werte in den Registern 0x02 bis 0x09 gesetzt werden.

Eine zeitliche Dezimierung wird bei uns nicht umgesetzt.

3.5 Datenausgabe

Bei voller Auflösung steht uns vorerst ein Datenstrom mit 864X625x16bit/Pixel (inkl. aller Blanking-Intervalle) zur Verfügung. Dies entspricht einer Auflösung von 720x576 aktiven Bildpunkten. Durch Setzen der entsprechenden Scaling- und Cropping-Register wird von uns momentan die Pixelausgabe auf 640x480 aktive Pixel eingeschränkt, um das Abspeichern eines kompletten Bildes im FIFO zu ermöglichen. Dabei wird nicht der eigentliche Datenstrom verändert, sondern nur das ACTIVE-Signal entsprechend angepasst.
Bei dieser Auflösung ist es nicht möglich, beide Halbbilder des Videosignals als ein Vollbild auszugeben. Sie werden jeweils aufeinanderfolgend als 240x640Pixel-Bilder ausgegeben. Welches der Halbbilder gerade ausgegeben wird, wird durch FIELD angezeigt. Synchronisationsinformationen werden bei uns nicht in den Datenstrom integriert.

Das Output-Interface unterstützt grundsätzlich zwei verschiedene Formate. Zum einen einen 16bit-Pixeldatenstrom (CLKx1) wobei jeweils ein Byte für das Helligkeits- und ein Byte für das Farbsignal (abwechseln U und V) stehen., zum anderen einen

8bit Pixeldatenstrom (CLKx2) wo Helligkeits- und Farbsignal byteweise zeitlich aufeinanderfolgen.
Bei uns kommt das erste Format zum Einsatz, welches im OFORM-Register festgelegt wird. Von uns werden dort die Standardwerte gesetzt, was einen von Steuerinformationen freien 16bit-Datenfluss zur folge hat.

Die 2 Bytes des Videodatenstroms werden auf 2 getrennten FIFO-Bausteinen geschrieben. Für uns ist hierbei nur das Byte/Pixel des Helligkeitssignals von Interesse. Über das Signal VRESET (Bildwechsel) wird der FIFO zurückgesetzt und bei als aktiv festgelegte Pixeln mit den Videodaten beschrieben indem Write_Enable des FIFOs über das ACTIVE Signal des Videochips aktiviert wird. Zur Synchronisation liegen dort die Signale von CLKx1 an. Hierzu mussten wir die zuvor auf CLKx2 kontaktierte Verbindung auftrennen und entsprechend auf CLKx1 legen.

Zu Beginn hatten wir leider sehr große Schwierigkeiten, ordentlich synchronisierte Bilder in die FIFOs zu schreiben. Es hat uns eine Menge Zeit gekostet, bis wir nun einige Bildwechsel lang die FIELD-Variable überprüfen, um eine korrekte Anordnung der Halbbilder und damit eine zuverlässige Synchronisierung des Datenstroms zu den tatsächlichen Halbbildwechseln zu erreichen.

3.6 Anmerkung

Eine komplette Beschreibung der Register und ihrer Adressen ist besser dem sehr umfangreichen Datenblatt zu entnehmen. Ebenso findet man dort noch genauere Blockdiagramme.

Der Schaltplan des Boards und das Datenblatt des Videochips sind in der Software-Zusammenstellung enthalten.

4. Iris-Scan Software

Die Software lässt sich in zwei Teile teilen, der eine Teil ist für die Steuerung der Karte zuständig, der andere für die Übertragung und Konvertierung der Bilddaten.

4.1 Die Steuerung der Karte

Der BT829 wird über eine I^2C Schnittstelle angesprochen, dazu stellt die Karte einen Port zur Verfügung, auf dem sowohl das Taktsignal, als auch die Daten an einer Adresse auf dem Bus des DSPs eingeblendet werden.
Dadurch können jetzt direkt Befehle an den Framegrabber gesendet werden, der sich auf eben diesem Daten- & Adressbus befindet.

Zur Initialisierung werden verschiedene Register auf dem Chip gesetzt, mit denen z.B. die Auflösung oder das Timing gesetzt werden.
Wir tasten mit der vollen PAL-Auflösung ab, schneiden aber nur ein Teil der Größe 640 x 480 Pixel aus dem Bild heraus, da die FIFOs auf der Karte nicht genügend Speicher für ein Vollbild zur Verfügung stellen.

4.2 Übertragung und Konvertierung der Bilddaten

Nach der Initialisierung schreibt der BT829 jedes Halbbild auf dem Bus raus.
Über einen weiteren Port können die FIFOs vom DSP aus ein-, ausgeschaltet und zurückgesetzt werden. Dadurch haben wir jetzt die Möglichkeit, immer zwei zusammengehörende Halbbilder in die FIFOs einzulesen.

Im nächsten Schritt werden die Bilddaten vom DSP wieder aus den FIFOs ausgelesen und die beiden Halbbilder, zu einem Vollbild zusammengesetzt, in einen Puffer geschrieben.

Das Bild besteht jetzt aus 640 x 480 Bildpunkten, jeder Bildpunkt wird mit einem 8-Bit Grauwert beschrieben. Alle Bildpunkte stehen in einem Array of Byte hintereinander.

4.3 Bildskalierung

4.3.1 Einleitung

Für einige der folgenden Bildverarbeitungsalgorithmen ist es von Vorteil, auf einer verkleinerten Version des Augenbildes zu arbeiten. Dadurch kann bei bestimmten Verarbeitungsschritten eine höhere Rechengeschwindigkeit erreicht werden und der Einsatz einiger Bibliotheksfunktionen wird dadurch erst sinnvoll. Da von unserer Seite keine hohen Anforderungen an die Flexibilität gestellt werden, reichte es aus, eine fest eingestellte Skalierung umzusetzen.

4.3.2 Ursprung

Als sich die Notwendigkeit ergab, eine schnelle Schärfebestimmung des Augenbildes durchzuführen, zwang uns die bisherige Erfahrung mit dem DSP auf schnelle Bibliotheksfunktionen zurückzugreifen.

Im speziellen ist hier die Umsetzung einer Diskreten Kosinustransformation gemeint, auf die im Absatz zur Schärfebestimmung eingegangen wird. Da diese Funktion mit festen Blockgrößen arbeitet, musste das Bild entsprechend den Gegebenheiten an die Funktion angepasst werden. Dies musste so geschehen, dass die relevanten Strukturen des Bildes erhalten bleiben, diese jedoch auch in den relativ kleinen DCT-Blöcken repräsentiert werden können.

Spätere Einsatzgebiete ergaben sich dann aus der Optimierung der Iris-Lokalisation, auf die ebenfalls an entsprechender Stelle näher eingegangen wird.

4.3.3 Die Skalierungsfunktion (*scale.c*)

Aufgrund der bereits angedeuteten Kriterien wurde von uns eine Bildskalierung von 640*480 (=300 kByte) auf 160*120 (=18,75kByte) Bildpunkte implementiert, was einer Verkleinerung auf 1/16 der Originalbildgröße entspricht.

Um den Rechenaufwand nicht unnötigerweise in die Höhe zu treiben und um Fehler durch Unterabtastung zu verringern, kam eine einfache blockweise Mittelwertfilterung zu Einsatz.

Hierbei wird ein 4*4 Block des Originalbildes durch einen Bildpunkt mit dem Mittelwert der 16 Ausgangbildpunkte ersetzt. Aliasing-Fehler sind natürlich trotzdem vorhanden, da die Filterung nicht überlappend arbeitet. Für unsere Zwecke ist diese Lösung jedoch völlig ausreichend.

Vorteilhaft ist vor allem das Herausfiltern hochfrequenter Störungen, die z.B. die Schärfemessung stören würden.

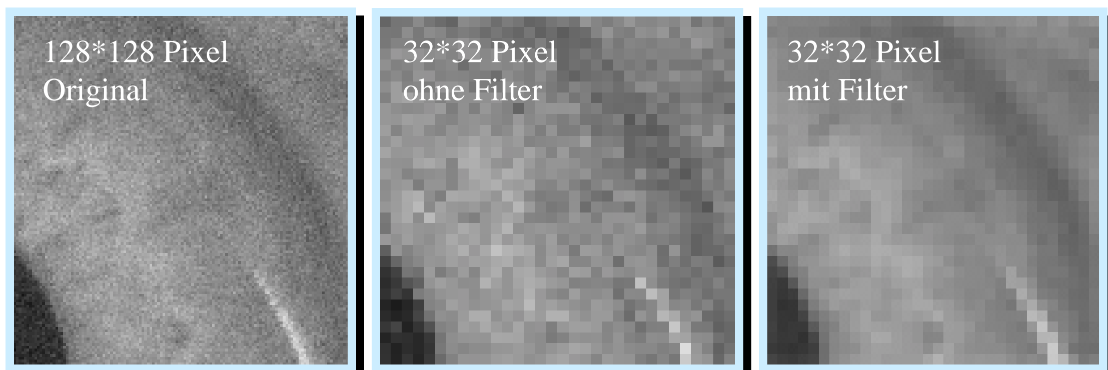


Bild 4.3.3.1 Veranschaulichung der Reduktion hochfrequenter Störungen durch Mittelwertfilterung

Die Zuordnung sieht dann folgendermaßen aus:

$$S(sx,sy)=\frac{1}{16}\sum_{dx=0}^{3}\sum_{dy=0}^{3}Bild\left(\ 4sx+dx,\ 4sy+dy\ \right)$$

Der Aufruf erfolgt mit folgender Funktion:

void
scale(unsigned char *buffer, unsigned char* scaled_buffer)

Außer den Speicherbereichen für Originalbild und die verkleinerte Version werden keine weiteren Parameter übergeben, was die Flexibilität stark einschränkt, jedoch möglicherweise komplizierte Umrechnungsroutinen vermeidet (z.B. nicht ganzzahlig umzusetzende Skalierungsverhältnisse).

Die Skalierung wird einmal für jedes aufgenommene Bild durchgeführt und kann in folgenden Verarbeitungsschritten weiterverwendet werden. Da relativ wenig Speicher benötigt wird, ist es auch nicht zwingend erforderlich, diesen an expliziter Stelle wieder freizugeben.

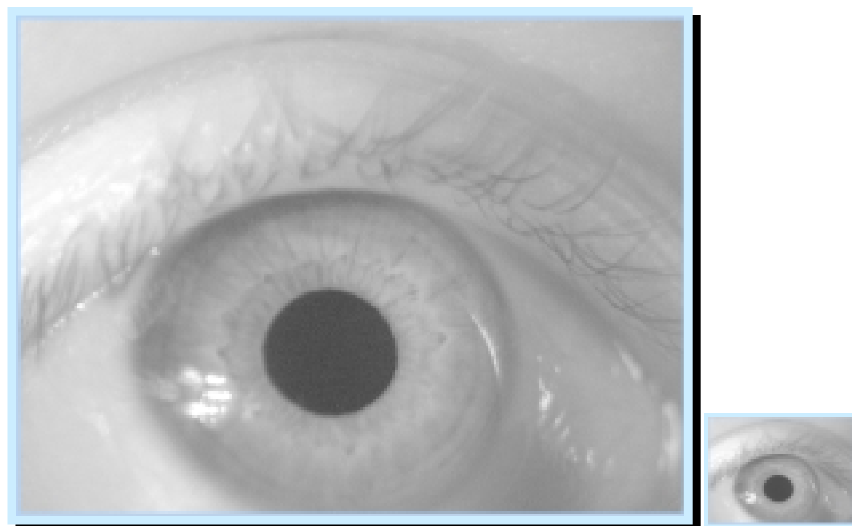


Bild 4.3.3.2 Größenvergleich von Originalbild und 1/16 verkleinertem Bild

4.3.4 Anmerkungen

Der Algorithmus arbeitet den Umständen entsprechend schnell. Eine Berechnung dauert im Normalfall weniger als 1 Sekunde. Dies erschien uns ausreichend, da im Normalfall nur ein Aufruf pro Iris-Code-Gewinnung nötig ist.

Bedingt ist die Rechenzeit vor allem dadurch, dass das komplette Originalbild durchlaufen werden muss (immerhin mehr als 300.000 Bildpunkte).

Zwei als entsprechend schnell angepriesene Bibliotheksfunktionen (***scale_vert*** und ***scale_horz***) sind zwar vorhanden, sie sind in diesem Fall sogar wesentlich flexibler,

jedoch sind sie für unser einfaches Anliegen zu komplex. Eine Skalierung kann nur jeweils horizontal oder vertikal durchgeführt werden, die Datenformate passen nicht zu unserer Anwendung (doppelter Speicherbedarf pro Bild) und es müssten extra entsprechende Filtermatrizen erzeugt werden.

4.4 Bildschärfe

4.4.1 Einleitung

Damit eine Weiterverarbeitung der Bilddaten überhaupt Sinn macht, ist es natürlich wichtig, dass die für uns relevanten Strukturen im Augenbild erkennbar sind. Ein wichtiges Kriterium hierfür ist die Bildschärfe.

Da die Kamera nur in sehr engen Grenzen fokussiert, kann es leicht passieren, dass eine unscharfes Bild durch leichte Kopfbewegungen bei der visuellen Überprüfung am Bildschirm unbemerkt bleibt.

Es war also eine technische Lösung für dieses Problem zu finden, die Auskunft über die Bildschärfe des tatsächlich aufgenommenen Frames liefert.

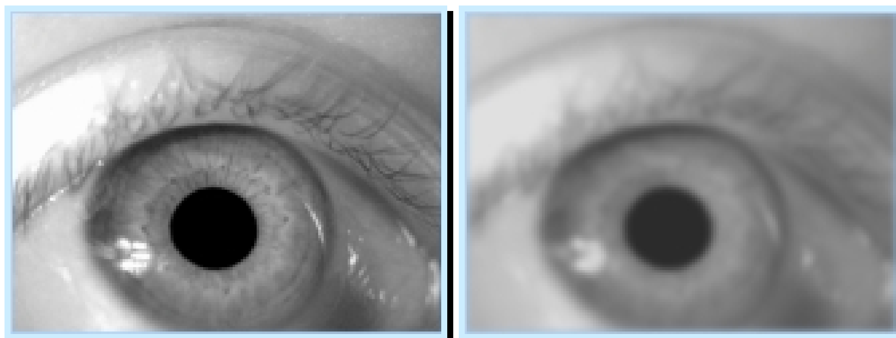


Bild 4.4.1 Scharfes Bild (Iris-Muster gut zu erkennen)/
unscharfes Bild (Iris-Muster nicht erkennbar)

4.4.2 Vorbetrachtung

Wäre die Helligkeitsverteilung des aufgenommenen Bildes bekannt, so könnte ein unkomplizierter Ansatz der örtlichen Ableitung in Betracht gezogen werden. Hierbei summiert man dann einfach die Beträge der örtlichen Ableitungen auf und gewinnt somit ein Maß zur Schärfebestimmung.

Da jedoch der Bildkontrast einen sehr hohen Einfluss auf das Ergebnis hat, ergibt sich nur eine ungenügende Unterscheidbarkeit von unscharfem, stark kontrastiertem Bild und scharfem, eher mattem Bild. Die Augenfarbe würde beispielsweise einen sehr hohen Einfluss auf das Resultat haben.

Eine statistische Analyse der Bildhelligkeitsverteilung und eine entsprechende nichtlineare Normalisierung könnten hierbei Abhilfe schaffen, würden jedoch auch einen unverhältnismäßigen Mehraufwand verursachen. Zudem würden hochfrequente Störungen, wie sie durch Bildrauschen entstehen, eine verheerende Verfälschung des Ergebnisses bewirken. Eine entsprechende Filterung wäre also ebenfalls erforderlich.

Da dieser Ansatz auch in professionellen Anwendungen nicht verfolgt wird, wurde er von uns auch schnell verworfen.

Eine weitaus effektivere Lösung bietet die Analyse des Amplitudendichtespektrums einer örtlich frequenzabhängigen Transformation. Da sich ein scharfes Bild vor allem durch den Gehalt an hohen Frequenzen auszeichnet, können über einen Vergleich der Transformationskoeffizienten leicht Aussagen über die Bildschärfe gemacht werden.

Natürlich bleibt auch hier eine Abhängigkeit vom Bildkontrast bestehen, sie wirkt sich aber nicht so gravierend aus. Zudem kann man hochfrequente Störungen durch Ignorieren der entsprechenden Transformationskoeffizienten vernachlässigen. Es werden also nur jene Ortsfrequenzen betrachtet, die die entscheidenden Strukturen repräsentieren.

4.4.3 Umsetzung

Von John Daugman wird eine zweidimensionale FFT über das gesamte aufgenommene Bild vorgeschlagen, um dann die Intensität der hochfrequenten Anteile über das Betragsspektrum zu bestimmen.

Ein entsprechender Algorithmus wäre sicherlich nicht schwer zu implementieren gewesen, jedoch hätte sich ein vergleichsweise hoher Rechenaufwand ergeben, bedingt vor allem durch viele Fließkommaoperationen in nicht optimalem Code. Da wir jedoch eine schnelle Entscheidung brauchen, um nötigenfalls eine erneute Bildaufnahme durchzuführen, kam diese Variante für uns nicht infrage.

4.4.4 Diskrete Kosinustransformation

Vom Prinzip her der FFT äußerst ähnlich arbeitet die Diskrete Kosinustransformation (DCT), zu der eine Bibliotheksfunktion (***fdct_8x8*** aus ***img62x.lib***) existiert. Da die Bildschärfebestimmung nur einen sehr kleinen Teil unseres Projekts ausmacht, haben wir uns für diesen, zunächst recht einfach erscheinenden Ansatz entschieden.

Die Zuordnungsvorschrift für die DCT sieht folgendermaßen aus:

$$\mathrm{DCT}(k,l) = \frac{2\cdot\alpha(k)\cdot\alpha(l)}{N}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}\mathrm{Img}(m,n)\cdot\cos\left(k\cdot\pi\cdot\frac{(2m+1)}{2N}\right)\cdot\cos\left(l\cdot\pi\cdot\frac{(2n+1)}{2N}\right)$$

$$\text{mit}\quad \alpha(z=0)=\frac{1}{\sqrt{2}}\quad\text{und}\quad \alpha(z\neq 0)=1$$

Das Bild selbst (inverse Transformation) stellt sich hierbei als gewichtete Summe von DCT- Basisbildern dar. Die Basisbilder bestehen jeweils aus kosinusförmigen Helligkeitsverläufen mit der horizontalen Frequenz k und der vertikalen Frequenz l.

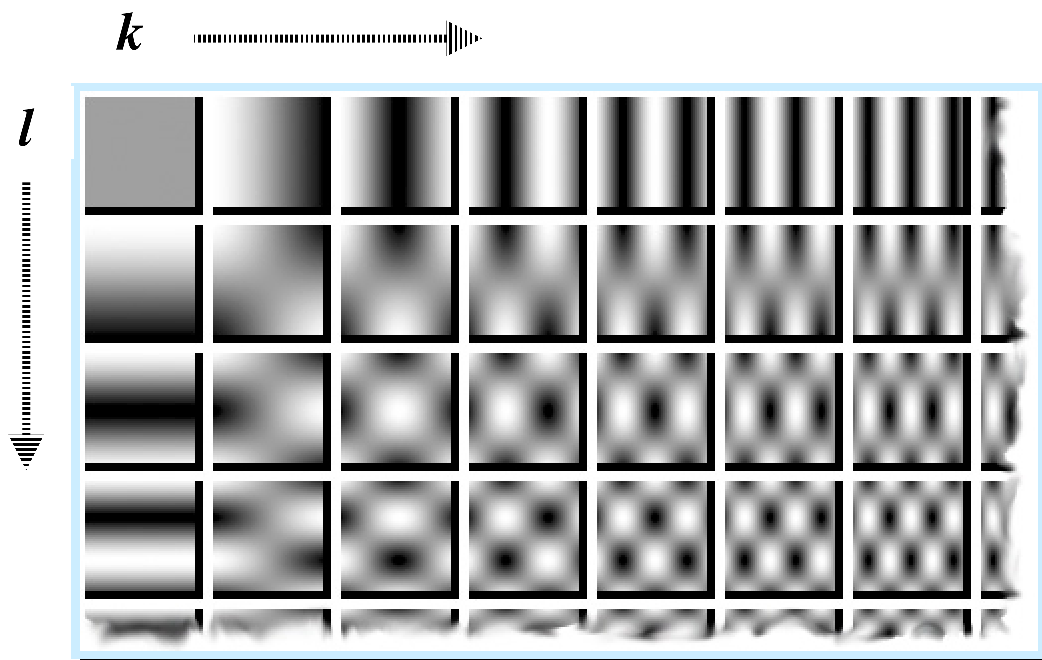


Bild 4.4.4 Ausschnitt aus der Matrix von DCT- Basisbildern für N=64

Die Gewichte dieser mittelwertfreien Basisbilder sind dann die Transformationskoeffizienten. Sie geben Auskunft über die Stärke der entsprechenden Strukturen.

Die Bibliotheksfunktion benutzt für die Berechnung der Transformationskoeffizienten den schnellen Chen-Algorithmus. Trotz herausragender Geschwindigkeit ist diese Funktion jedoch wieder beispielgebend für inflexible Implementierungen. Da die Einstellung entscheidender Parameter nicht möglich ist, mussten wir die Eingangsdaten entsprechend unseren Anforderungen anpassen.

4.4.5 Datenanpassung

Die Funktion ***fdct_8x8*** erwartet als Bilddatenargumente ***short***-Werte, was eine Umwandlung unseres nur halb so großen Formats ***unsigned char*** erfordert. Allein dies

würde bei einer Originalbildgröße von 640*480 schon eine erhebliche Speicherbelastung bedeuten (zusätzliche 600kByte).
Zudem müssen die Bilddaten für sinnvolle Ergebnisse in einem Vektor von n Blöcken mit N*N zusammenhängenden Bildpunkten übergeben werden. Dies nötigt uns, eine komplette örtliche Umstrukturierung des Bildes vorzunehmen, die folgendermaßen aussieht:

$$Daten\left(n,\Delta x,\Delta y\right)=Bild\begin{pmatrix} 8 \cdot \left(n \mod \dfrac{Bildbreite}{N}\right) + \Delta x, & 0 \le \Delta x \le N-1, \\ 8 \cdot \left(n \operatorname{div} \dfrac{Bildbreite}{N}\right) + \Delta y & 0 \le \Delta y \le N-1 \end{pmatrix}$$

Die entsprechende Zuweisung der linearen Datenpuffer sieht natürlich ein wenig anders aus. Sie ist aus dem beigefügten Quelltext ersichtlich.

Das gravierendste Problem ist jedoch die Blockgröße, die für diese Funktion auf **N=8** festgelegt ist. Das folgende Bild veranschaulicht, dass bei dieser Blockgröße eine Unterscheidung nach der Erkennbarkeit von relevanten Strukturen kaum möglich ist.

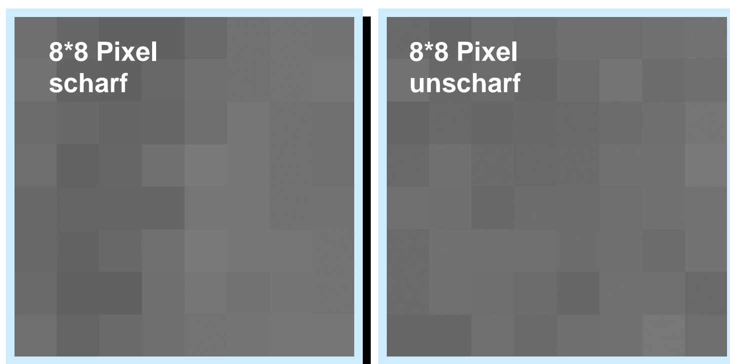


Bild 4.4.5.1 Blockvergleich bei Originalbildgröße

Den weitaus größten Anteil der höheren Frequenzen bildet wahrscheinlich ein leichtes Bildrauschen, welches sich nie ganz ausschließen lässt. Eine entsprechende Filterung könnte diese zwar beseitigen, die Auflösung im interessanten, hier tieffrequenten Bereich bleibt jedoch unzureichend. Tests mit verschiedenen scharfen und unscharfen Bildern führten zu keiner eindeutigen Schärfegrenze, was eine Änderung der Vorgehensweise nötig machte.

Um also die Transformation in für uns nützlichen Dimensionen durchzuführen, wird von uns eine vorhergehende Skalierung des Augenbildes auf 1/16 der Originalgröße (= 160*120 Bildpunkte) durchgeführt. Grundlage für die Größenverhältnisse waren visuelle Tests an unterschiedlich verkleinerten Augenbildern. Die hierzu verwendete Funktion wird im entsprechenden Abschnitt ***Bildskalierung*** beschrieben. Der Aufruf erfolgt außerhalb der Schärfebestimmung, da das verkleinerte Bild auch noch in anderen Verarbeitungsprozessen verwendet wird.

Wie man in folgender Abbildung sehen kann, treten nun die schärferelevanten Strukturen in den kleinen Blöcken deutlicher hervor und Störungen haben kaum noch Anteil an der Transformation.

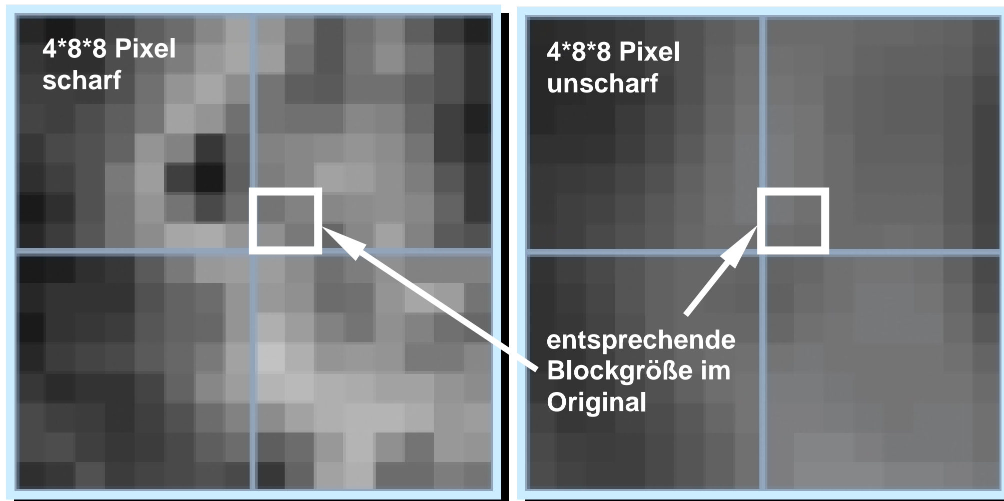


Bild 4.4.5.2 Blockvergleich bei verkleinertem Bild

Auf dem skalierten Bild wird nun zunächst die schon beschriebene Datenumstrukturierung vorgenommen. Um die Schärfe vor allem im Bereich der Iris zu messen, wird dabei nur ein zentraler Bereich berücksichtigt. Von den 160*120 Bildpunkten werden nur die mittleren 80*64 Bildpunkte verarbeitet. Geschwindigkeitsüberlegungen spielten hierbei keine Rolle. Die so entstehende Kette von 80 Bildblöcken wird dann an die DCT-Funktion übergeben:

```
void
fdct_8x8(short * dct_data, unsigned num_dcts)
```

Es werden also die aufbereiteten Bilddaten und die Anzahl der Bildblöcke übergeben. Die Funktion arbeitet direkt auf den Daten, so dass nach dem Aufruf in ebendiesen Blöcken die Transformationskoeffizienten zu finden sind. Diese sind im folgenden Bild (nach entsprechender Umwandlung in Bildblöcke) dargestellt.

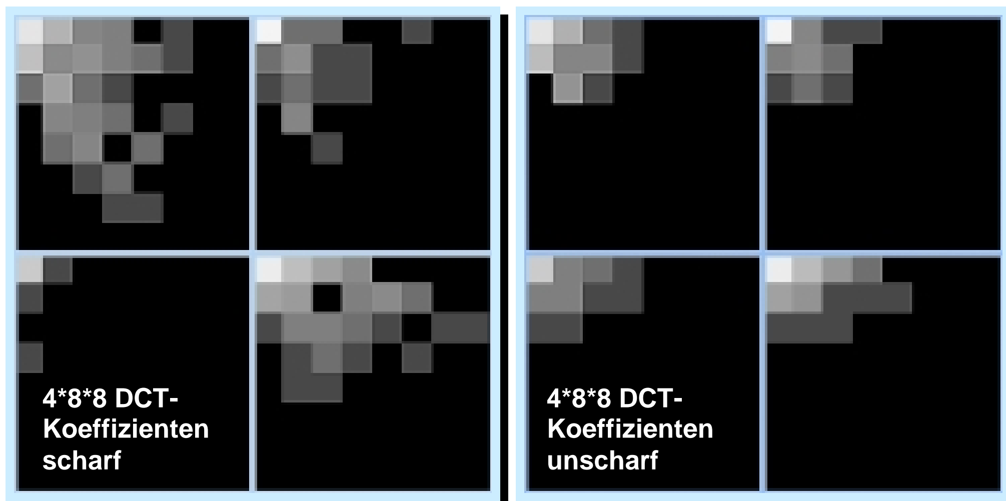


Bild 4.4.5.3 Vergleich der DCT- Koeffizientenmatrizen (Beträge)

Sehr deutlich sind hier die Unterschiede zwischen den Koeffizientenmatrizen des scharfen und des unscharfen Bildes zu sehen. Für das scharfe Bild sind die Koeffizienten höherer Ordnung wesentlich stärker ausgeprägt, was auf das Vorhandensein von entsprechend höherfrequenten Strukturen hinweist.

4.4.6 Auswertung der Transformationsdaten

Um einen verlässlichen Wert für die Bildschärfe zu erhalten, werden die Beträge der relevanten Transformationskoeffizienten aufsummiert, Phaseninformationen sind ja nicht von Belang. Der optimale Bereich hierfür wurde durch Tests ermittelt, die zum Ziel hatten, eine bestmögliche Unterscheidbarkeit von scharfen und unscharfen Bildern zu liefern. Die Summation wird dann über alle 80 berechneten Transformationsmatrizen ausgeführt. Folgendes Bild zeigt den von uns verwendeten Koeffizientenbereich:

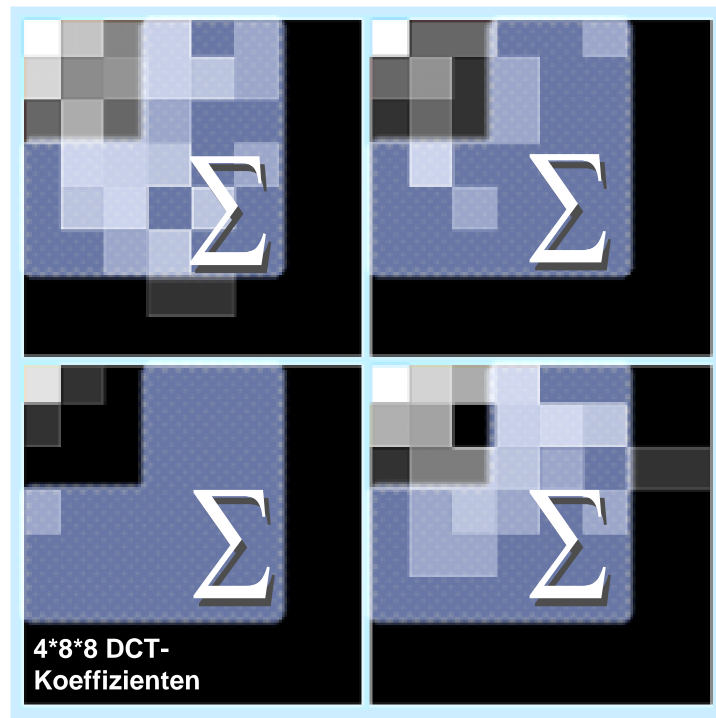


Bild 4.4.6 Summationsbereiche in den DCT- Koeffizientenmatrizen

Die niederfrequenten Anteile werden selbstverständlich nicht in die Auswertung mit einbezogen. Die sehr hochfrequenten Bereiche werden ebenfalls ausgelassen, da die entsprechenden Beträge sehr klein sind und sich eher in den Bereich von möglichen Bildstörungen einordnen lassen.

4.4.7 Die Schärfefunktion (*sharpness.c*)

Der bisher beschriebene Ablauf wird in einem Funktionsaufruf zusammengefasst.

int
sharpness(unsigned char * scaled_buffer)

Das verkleinerte Augenbild wird zuvor erzeugt und dann an die Funktion übergeben.

Innerhalb der Funktion wird dann die Datenumstrukturierung vorgenommen und die Transformation mit der Bibliotheksfunktion vorgenommen.
Dann folgt die partielle Summation über die DCT-Blöcke (2 < x,y < 6).
Der erhaltene Summenbetrag wird als Integer-Wert zurückgegeben, um eine externe Interpretation zu ermöglichen.

Momentan sind Summenrückgabewerte bis ***shaerfe*** = 12.500 aufgetreten und ein Schwellwert von ***sharpness_threshold*** = 5.500 liefert ein sehr zuverlässiges Entscheidungskriterium. Liegt die Summe unterhalb dieses Grenzwertes ist das Bild unscharf und muss erneut aufgenommen werden.

4.4.8 Anmerkungen

Durch unsere Anpassungen arbeitet die Funktion sehr zuverlässig. Auch helle Augenfarben stören die Entscheidung nicht.

Die Transformation selbst (Bibliotheksfunktion) benötigt wegen der guten Optimierung sehr wenig Rechenzeit:

Anzahl der Zyklen = 48 + 160*Blockanzahl = 48 + 160*80 = 12848

Der Großteil der Rechenzeit geht auf das Konto der Datenumstrukturierung und der Summation der Koeffizientenbeträge. Da wir jedoch durch die entsprechenden Skalierungen und Bereichseinschränkungen die Datenmenge sehr klein halten, kann der ganze Funktionsaufruf in wenigen Millisekunden abgearbeitet werden. Wesentlich länger dauert die vorangehende Bildskalierung.

Die Schärfeschwelle lässt sich leicht im Hauptprogramm einstellen und ermöglicht damit eine flexible Anpassung an Veränderungen der Aufnahmehardware (Lichtverhältnisse etc.) oder der Videochip-Einstellungen (Kontrast etc.).

4.5 Irislokalisierung

4.5.1 Einleitung

Um die Iris zu analysieren, muss sie zunächst aus dem Gesamtbild ausgeschnitten werden. Dazu muss

a) als erstes die Iris im Bild lokalisiert werden.

b) Und außerdem ein Verfahren gefunden werden, durch das sich die Iris vom restlichen Bildmaterial trennen lässt. Das impliziert sowohl das Trennen von den Flächen außerhalb der Iris, als auch das Entfernen der Pupille innerhalb der Iris.

Wir wählten den im Folgenden beschriebenen Ansatz um die genannte Aufgabe zu erledigen (1).
Andere Ansäte gingen voraus, wurde von uns allerdings nach einigem Probieren (Simulationen mit MATLAB) verworfen. Mehr Information zu den anfänglichen Herangehensweisen findet sich unter Pkt.4.5.4 .

Aber zunächst der aktuelle Ansatz:

4.5.2 Ansatz

4.5.2.1 Einleitung

Wegen des großen Aufwands der anfänglich verfolgten Herangehensweise (s. Pkt.4.5.4) haben wir uns entschlossen, eine einfachere, schnellere Methode für die Lokalisierung der Iris zu finden und zu implementieren. Der Algorithmus, den wir im Folgenden beschreiben, ist schneller, und hat bei der Mehrheit der Testbilder gut funktioniert.
Um eine korrekte Lokalisierung zu bekommen, sind von den Bildern folgende Eigenschaften erwünscht:

- Möglichst viel Kontrast zwischen Pupille und Iris
- Möglichst viel Kontrast zwischen Iris und dem Rest des Auges
- Möglichst wenige Lichtreflexionen im Bereich der Pupille und der Iris

Der Algorithmus beruht auf die Tatsache, dass man nur drei Punkte braucht, um einen Kreis vollständig zu bestimmen. Wir haben den Prozess in verschiedenen Modul-Funktionen gegliedert. Das ganze Verfahren, und die Beziehungen zwischen den Modulen, werden im folgenden Schema gezeigt:

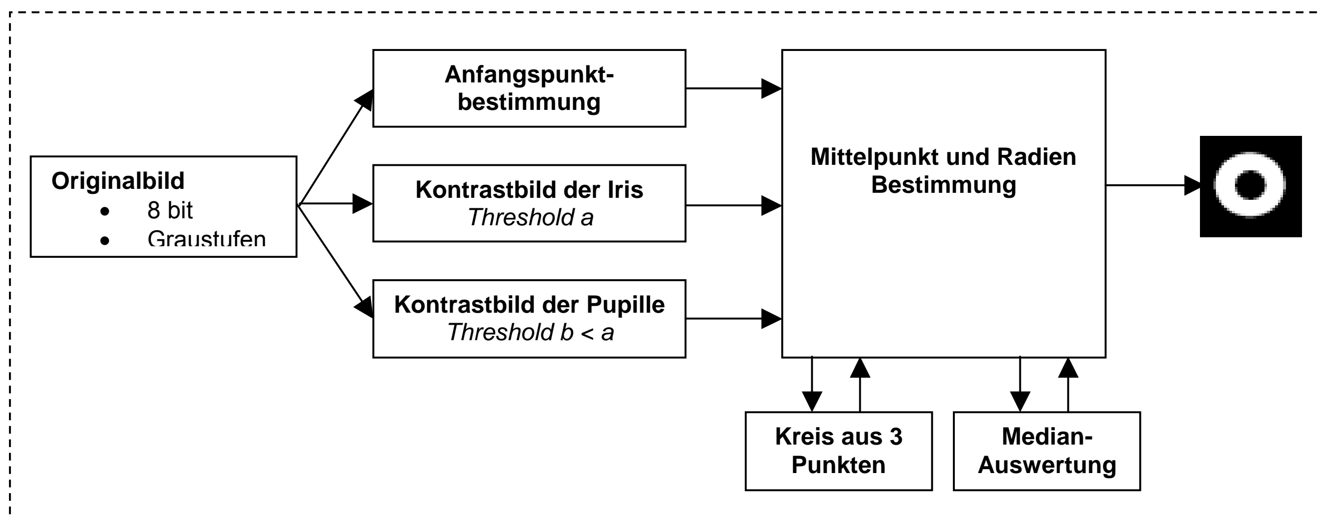


Bild 4.5.2.1 Schema der Iris-Lokalisierung

Der ganze Prozess bekommt als Input das Originalbild (640x480 Pixels, 1 Byte pro Pixel, Graustufen), und übergibt als Output die Koordinaten des Mittelpunktes der Pupille sowie die Radien der Pupille und der Iris. Wir können annehmen, dass Pupille und Iris genau konzentrisch sind, also brauchen wir nur einen von den beiden Mittelpunkten zu übergeben.

Die Anfangspunktbestimmung ist in der Datei *locate.c* implementiert. Die Bestimmung des Mittelpunktes und der Radien, und die dazugehörigen Funktionen (z.B. die Bestimmung eines Kreises aus drei Punkten und die Median-Auswertung) sind in der Datei *iscan.c* implementiert (siehe Quellcode).

4.5.2.2 Anfangspunktbestimmung (*locate.c*)

Damit der Algorithmus zur genauen Bestimmung der Iris-Dimensionen greifen kann, muss zuerst ein Bildpunkt innerhalb der Pupille gefunden werden. Hierzu kommt eine Korrelationsgradmessung zwischen Augenbild und einer idealisierten Pupille zum Einsatz.

Idealisierung einer Pupille

Zunächst wurden von uns Größenbestimmungen anhand von Testbildern durchgeführt. Man kann leicht erkennen, dass sich die Pupille bei einer guten Fokussierung als schwarzer, kreisförmiger Bereich vom restlichen Bild abhebt. Auch bei unterschiedlicher Pupillenweitung können relativ gute Voraussagen über ihre räumlichen Ausdehnungen gemacht werden.
Die entsprechenden Skalierungen sind in der Header-Datei ***locate.h*** sehr einfach einzustellen.

Pupillen-Radius ≈ 1/8 der vorliegenden Bildbreite

Zur Umsetzung dieser Idealpupille wird nun ein Graustufenbild erzeugt, dass den gesamten Bereich einer Pupille in entsprechender Relation aufnehmen kann.

Pupillen-Bild = (2*Pupillen-Radius)² Bildpunkte

Da aufgrund unterschiedlicher Pupillenweitung kleinere Unterschiede in der Ausdehnung auftreten, wurde von uns ein eher unscharfes Idealbild erzeugt. Dennoch sollte natürlich die Kreisform erhalten bleiben. Für die Helligkeit der Pupillenbildpunkte wurde von uns daher folgende Zuordnung gewählt:

$$Pupille(x,y) = \left(\frac{x^2+y^2}{2\cdot Pupillenradius^2}\right)^2 \qquad , -r \le x,y \le +r$$

Der Funktionsaufruf erfolgt einmal für die ganze Suche mit:

void
draw_eye (unsigned char * eye).

Das entstehende Bild (Koordinatenursprung in der Bildmitte) sieht dann folgendermaßen aus:

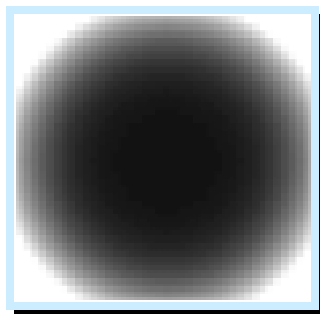


Bild 4.5.2.2.1 Idealisierte Pupille

Maximale Korrelation

Um die Lage der Pupille zu ermitteln, wird nun der Ort der maximalen positiven Korrelation zwischen Augenbild und Idealpupille gesucht.

$$XY(Pupille) = XY\left(\max \left\{ \rho(x,y) = \sum_{\Delta x}^{Radius} \sum_{\Delta y}^{Radius} Bild\left(x \pm \Delta x,\ \ y \pm \Delta y\right) \cdot I.Pup\left(\pm \Delta x,\ \ \pm \Delta y\right) \right\} \right)$$

Die Umsetzung dieser Suche erfolgt mit dem Aufruf von:

struct point
max_corr (unsigned char * buffer, unsigned char * eye, struct point start, struct range search)

Es werden das Augenbild und die Idealpupille sowie Skalierungen des Suchbereichs übergeben (Startpunkt und Auflösung).

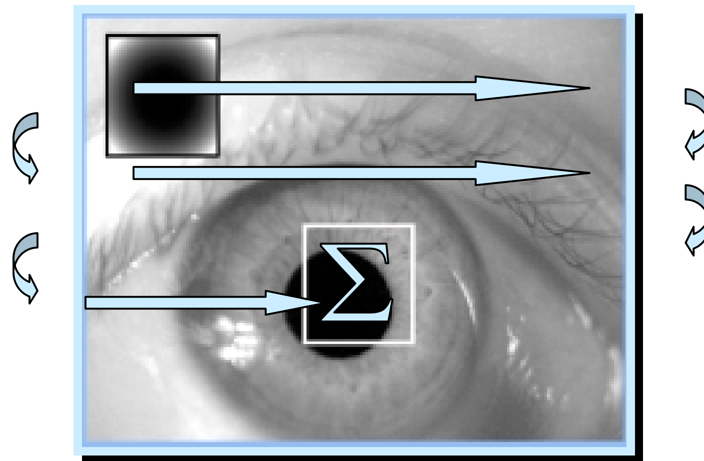


Bild 4.5.2.2.2 Kreuzkorrelation zwischen Augenbild und Idealpupille

Optimierung des Algorithmus

Während der Bestimmung der Korrelationssumme werden eine Vielzahl von Fließkomma-Operationen durchgeführt. Dies ist auf dem Festkomma - DSP eine sehr zeitintensive Angelegenheit. Daher wird eine iterative Wiederholung des Suchalgorithmus durchgeführt. Es wird mit einer großen Schrittweite und einer sehr groben Auflösung begonnen, die dann bei jedem weiteren Aufruf entsprechend verfeinert wird. Ist eine bestimmte Genauigkeit oder Anzahl von Funktionsaufrufen erreicht, wird die Suche abgebrochen und die erhaltenen Koordinaten werden als in der Pupille befindlich angenommen.

unsere Anforderungen (***locate.h):***

Resultatabweichung zwischen 2 Aufrufen (***precision***)	=	1/160 der Bildbreite

Maximalanzahl der Iterationen
(***max_depth***) = 4

Ermöglicht wird die Iteration durch die Übergabe von Startpunkt, Suchbereich und Auflösung an die Korrelationsfunktion. Die Übergabe der entsprechenden Werte erfolgt in zwei verschiedenen Strukturen (***struct point*** und ***struct range***), deren Elemente den Suchbereich durch Angabe von Start- und Endwerten und die Auflösung durch Einstellen der Inkremente steuern.

die Ausgangswerte dieser Parameter (***locate.h***):

Startpunkt der Suche (fest) = Bildmittelpunkt

Startbereich der Suche (fest) = ganzes Bild - Pupillenradius

Startschrittweite für untersuchte Ortskoordinaten
(***start_step*** für ***x+=*** und ***y+=***) = 1/32 der Bildbreite

Startauflösung der Korrelationssumme
(***start_res*** für ***Δx+=*** und ***Δy+=***) = 1/32 der Bildbreite

Bei jeder Iteration wird die Ausdehnung des Suchbereichs geviertelt, die Schrittweite der Summe halbiert und die Auflösung in der Summation verdoppelt. So wird in etwa eine konstante Rechenzeit für jeden Aufruf der Korrelationsfunktion erreicht. Es konnte bisher kein Fehlverhalten festgestellt werden, die Robustheit des Algorithmus bleibt also ebenfalls erhalten.

Zudem wird die Tatsache ausgenutzt, dass zuvor bereits eine verkleinerte Version des Augenbildes erzeugt wurde (Abschnitt ***Bildskalierung***). Der gesamte bisher beschriebene Ablauf erfolgt also auf diesem verkleinerten Bild, was eine geringere Datenmenge und demzufolge auch kürzere Rechenzeiten zur Folge hat.

nötige Voreinstellungen für Arbeit auf skaliertem Eingangsbild (***locate.h***):

Bildbreite (***H_SIZE***) = 160
Bildhöhe (***V_SIZE***) = 120

Die gesamte Anfangspunktbestimmung wird in folgender Funktion zusammengefasst. Sie wird innerhalb der Bildverarbeitungsfunktion ***GetScanCode(...)*** aufgerufen und schreibt die relevanten Werte zur Weiterverarbeitung in die entsprechenden Variablen (***mx_pup***, ***my_pup***) :

void
locate_pupil (unsigned char * buffer, int * mx_pup, int * my_pup);

Eingangsbild (***buffer***) = verkleinertes Augenbild (***scaled_buffer***)

Anmerkungen

Die von uns implementierte Variante des beschriebenen Algorithmus hat eine Rechenzeit von etwa 2-3 Sekunden. Bei einer Probemessung wurden ca. 780 Millionen Zyklen auf dem DSP gezählt.

Trotz einer Reduktion der anfänglichen Rechenzeit von ca. 20 Sekunden auf die genannten Werte bleibt ohne Frage noch ein erheblicher Bedarf an Optimierung bestehen.

Verursacht wird der Rechenaufwand vor allem durch jeweils 4 ineinandergeschachtelte Schleifen beim Aufruf der Korrelationsfunktion. Natürlich müsste der prinzipielle Ablauf erhalten bleiben, jedoch kommt es bisher noch zu keiner Ausnutzung der Pipeline-Eigenschaften beim Berechnen der Summenwerte und auch über eine Veränderung der Datenformate (Festkomma- statt Fließkomma-Berechnungen) ließe sich möglicherweise eine effizientere Lösung finden.

Letztendlich bleibt die Rechenzeit jedoch im erträglichen Bereich und die Sicherheit beim Auffinden der Pupille wurde in vielen Tests unter Beweis gestellt.

4.5.2.3 Kontrastbildern erstellen (Thresholding)

Das Erstellen der Kontrastbilder dient nicht direkt zur Findung der Radien oder des Mittelpunktes der Iris und Pupille. Es sollen jedoch die Bilder so aufbereiten werden, dass die eigentlichen Algorithmen leichter zum gewünschten Ziel gelangen.
Die Kontrastbilder werden durch ein Prozess namens „Thresholding“ erstellt. „Thresholding“ bedeutet:

Die Pixel, die einen Grauwert haben, der über einen bestimmten Grenzwert (Threshold) liegt, vom Rest der Pixel zu unterscheiden.

In unserem Fall wandeln wir die Pixel, die dunkler als eine bestimmte Graustufe sind (0=Schwarz, 255=weiß), in schwarze Pixel, der Rest bleibt unverändert.

Für das Thresholding benutzen wir die Funktion *threshold*, die in der Bildverarbeitungs-Library IMGLIB von Texas Instruments (Datei *img62x.lib*) vorhanden ist (3). Die Funktion bekommt als Input das Originalbild und den Grenzwert „pupille_thresh“ bzw. „iris_thresh“, und gibt als Output das Kontrastbild.

Die Threshold-Werte sind die wichtigsten Parameter bei der Iris-Lokalisierung. Um eine erfolgreiche Erkennung zu erreichen, muss man diese Werte sorgfältig wählen. Im Bild 4.5.2.3 wird einen Vergleich zwischen einem Originalbild und seinen entsprechenden Kontrastbildern für die Pupille und für die Iris gezeigt. Alle Beispielbilder im vorliegenden Endbericht wurden von uns aufgenommen, mit dem DSP verarbeitet, und mit dem Programm „DSPView“ angezeigt.

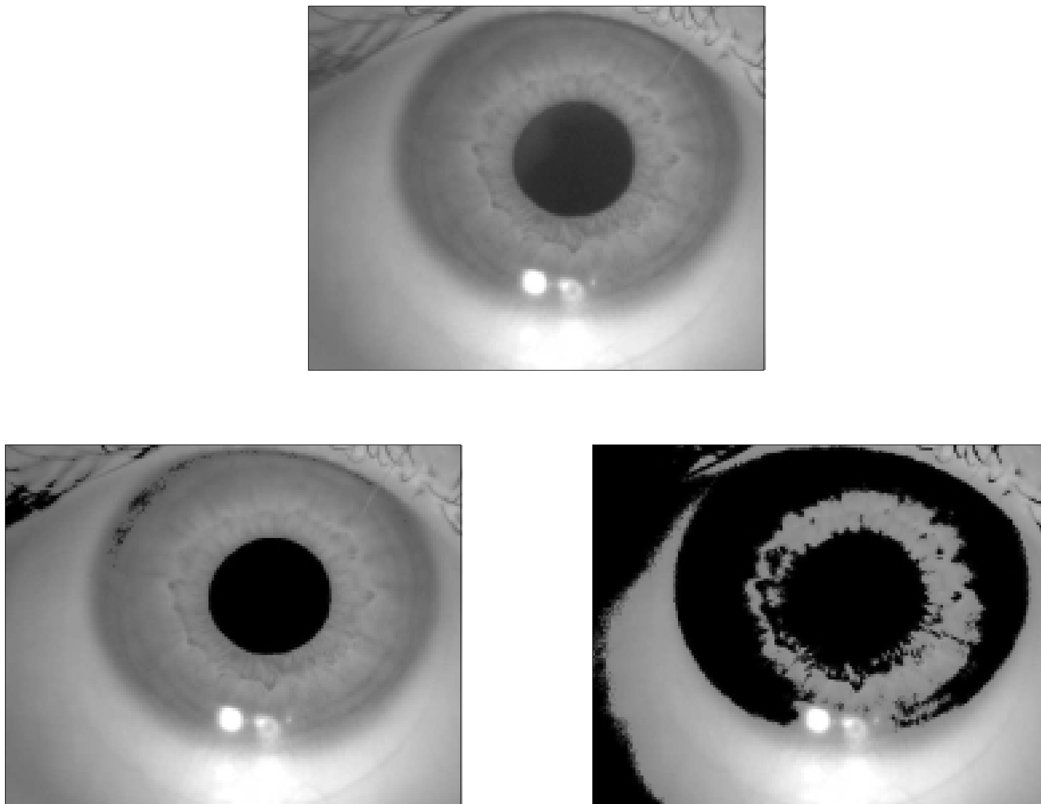


Bild 4.5.2.3 Originalbild (oben) mit Kontrastbildern der Pupille (unten links) und der Iris

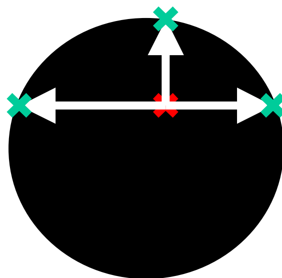
An diesem Beispiel kann man sehen, dass manche Teile in der Mitte der Iris heller als der äußere Bereich sein können; der folgende Algorithmus hat dies zu berücksichtigen. Wichtig ist aber, dass die ungefähr kreisförmigen Ränder in den Kontrastbildern gut erkennbar werden. Die Kontrastbilder werden nun an den „Drei-Punkte-Algorithmus“ übergeben.

4.5.2.4 Drei-Punkte-Algorithmus

Der Algorithmus, den im Folgenden erklärt wird, dient zur Bestimmung des *genauen* Mittelpunktes der Pupille bzw. Iris und der Radien der Kreise, die die Iris und die Pupille begrenzen.
Der Algorithmus wird zweimal durchgeführt: einmal für die Pupille, einmal für die Iris. Zur Erklärung betrachten wir zunächst ein ideales Kontrastbild einer Pupille (bzw. Iris): einen schwarzer Kreis.

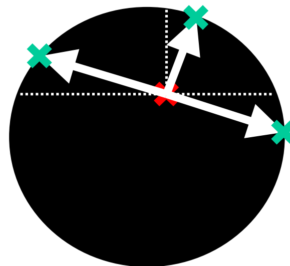
<u>Lokalisieren der Randpunkte</u>

Wir gehen folgendermaßen vor: Vom Anfangspunkt, den wir vorher bestimmt haben (siehe Abs. 4.5.2.2), lesen wir alle Pixel entlang drei zueinander senkrechten Linien ein, bis einen hellen Pixel erreicht wird (siehe Bild 4.5.2.4a). Die Koordinaten dieser Grenzpunkte werden gespeichert. Somit erhalten wir drei Punkte, die an die Funktion „Kreis" (s. weiter unten) übergeben werden, um den Mittelpunkt und den Radius des Kreises zu berechnen.



1

Bild 4.5.2.4a



2

Bild 4.5.2.4b

...
usw

3.-6.

Bild 4.5.2.4.1 Drei-Punkte-Algorithmus

Das würde im Idealfall genügen, doch in der Praxis treten verschiedene Ungenauigkeiten auf. Vor allem müssen wir folgendes betrachten:

- Pupille und Iris sind keine perfekten Kreise
- Unerwünschte Lichtreflexionen können in den Kontrastbildern weisse Flecken innerhalb der Iris oder der Pupille verursachen
- Einige Bereiche innerhalb der Iris können heller als der äussere Bereich des Auges sein

Unsere erste Version des Algorithmus zog diese Probleme nicht in Betracht (siehe MATLAB-Simulation im Zwischenbericht). Seitdem haben wir also das Programm reichlich erweitert, um damit umzugehen. Die drei erwähnte Probleme haben wir folgendermassen behandelt:

- Pupille und Iris sind nicht perfekte Kreise

Obwohl die drei Grenzpunkte korrekt erkannt worden sind, kann der berechnete Kreis vom tatsächlichem Rand der Pupille oder Iris(oft etwas elliptisch) abweichen.

Um eine bessere Näherung zu bekommen berechnen wir nicht nur ein, sondern sechs Kreise, indem wir jedes Mal die drei senkrechte Achsen etwas drehen (siehe Bild 4.5.2.4b) und die jeweiligen Grenzpunkten speichern. Danach werden die sechs erhaltene Mittelpunkte und Radien durch die „Median-Auswertung“ gemittelt (siehe Abs. Medianauswertung). Schematisch lautet jetzt unser Algorithmus also:

1. Wir lesen die Pixel auf drei zueinander senkrechten Achsen ein, bis ein heller Pixel erreicht wird
2. Die Koordinaten der drei Pixel an der Grenze werden gespeichert
3. Aus den 3 Koordinatenpaaren werden einen Mittelpunkt und einen Radius berechnet (Lösung eines LGS)
4. Wir drehen die Achsen. Zurück zu 1. (6 Mal)
5. Berechnung des Median-Wertes der 6 erhaltenen Mittelpunkten und Radien

- Lichtreflexionen

Die Infrarot-LEDs, die im Kameragehäuse für die Beleuchtung zuständig sind, verursachen unvermeidliche Reflexionen auf der Iris. Da wir aber wissen, dass diese immer am selben Ort auftreten, können wir diesen Bereich für die Codierung auslassen, wie später erklärt wird. Trotzdem können außerdem andere, unerwünschte Lichtreflexionen in anderen Bereichen des Auges auftreten, die, z.B. von der Raumbeleuchtung stark abhängen können. Dies kann zur Entstehung heller Flecken im Kontrastbild führen.

Um diese weißen Pixel nicht als Pixel an der Kreisgrenze zu betrachten, benutzen wir die Variable ´max´. Diese gibt an, wie viele zusammenliegende weiße Pixel man in einer Richtung einlesen muss, bevor man den ersten weißen Pixel in dieser Gruppe als Pixel auf dem Kreisrand erklärt (es entsteht also eine Art „Sicherheitsbereich“). Nach vielen Tests im Labor haben wir als geeigneten Wert max=25 gewählt. Bild 4.5.2.4.2 zeigt alle „Erkennungsachsen“ für die Erkennung einer Pupille. Die Sicherheitsbereiche sind deutlich zu Erkennen.

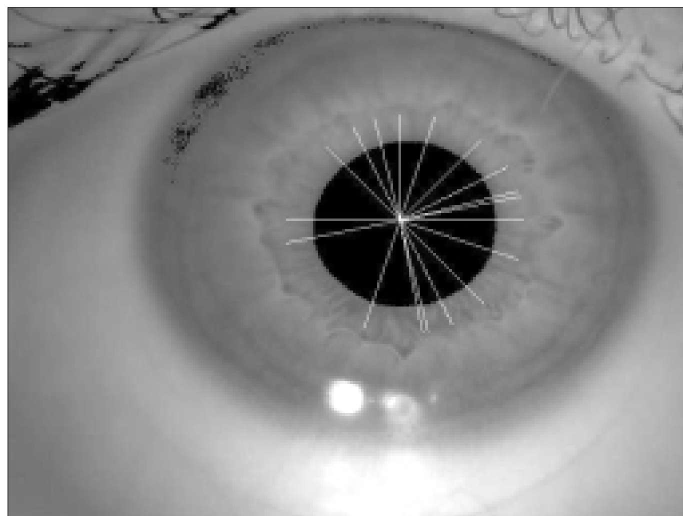


Bild 4.5.2.4.2 Pupille-Erkennung

- Helle Bereiche in der Iris

Manchmal sind in der Iris Stellen enthalten, die durch unsere Beleuchtungsmethode heller als der äußere Bereich des Auges erkannt werden können. Ähnlich wie bei Lichtreflexionen, entstehen dadurch im Kontrastbild helle, unerwünschte Bereiche. Jetzt handelt es sich aber nicht um kleine Flecken, sondern um größere, oft ringförmige Strukturen (Bild 4.5.2.4.3). Deshalb haben wir das Problem getrennt behandelt.

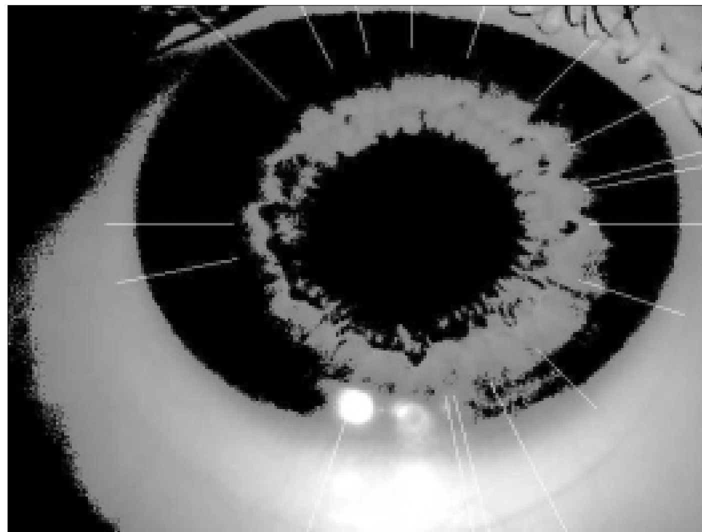


Bild 4.5.2.4.3 Iris-Erkennung

Wenn es sich um die Erkennung einer Iris handelt, beginnen wir mit dem Auswerten der Pixel *nicht* im Anfangspunkt.
Wir folgen den selben Linien wie bei der Bestimmung der Pupille, aber fangen mit der Überprüfung der Grauwerte erst an, wenn wir einen gewissen Abstand zum Anfangspunkt haben, der größer als zwei Mal der Radius der Pupille ist. Das verhindert, dass der erste dunkel-hell Übergang (also die innere Grenze der ringförmigen Struktur) als Irisrand erkannt wird.

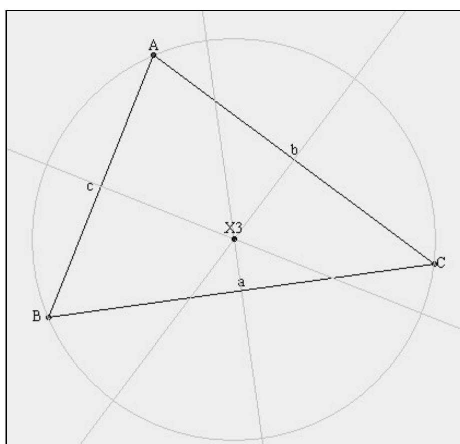


Bild 4.5.2.4.4 Umkreismittelpunkt

<u>Bestimmung eines Kreises aus drei Punkten</u>

Aus der Geometrie wissen wir, dass die drei Mittelsenkrechten eines Dreiecks sich im Umkreismittelpunkt schneiden (Bild 4.5.2.4.4). Man kann relativ leicht die Bestimmung des Mittelpunktes durch Vektorgeometrie programmieren (in diesem Fall sind die Koordinaten der Ebene einfach die Positionsindexe der Pixels in der Bildmatrix, (1,1) = oben, links). Man berechnet zuerst die Richtungsvektoren von zwei Seiten des Dreiecks, indem man die Koordinaten von je

zwei Punkten subtrahiert. Sei (a,b) der Richtungsvektor einer Seite, dann ist (b,-a) der Richtungsvektor der entsprechenden Mittelsenkrechten. Die Gleichungen der zwei Mittelsenkrechten werden ausgeglichen, und somit erhalten wir ein lineares Gleichungssystem, das mit der Cramerschen Regel gelöst wird.

Medianauswertung

Bei jeder Drehung der Erkennungsachsen werden die Koordinaten der drei vermutlichen Grenzpunkte in eine Matrize gespeichert (wir erhalten einen 6-stelligen Vektor von Radien, einen mit 6 X-Koordinaten, und einen mit 6 Y-Koordinaten). Die Medianauswertung dient dazu, aus den sechs gegebenen Kreisen einen endgültigen Kreis zu wählen, der den tatsächlichen Rand am besten trifft. Es gibt verschiedene Möglichkeiten, das zu realisieren. Wir haben probiert, einfach das arithmetisches Mittel von Radien, X-Koordinaten und Y-Koordinaten zu berechnen:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Nun wissen wir, dass wahrscheinlich die meisten Kreise korrekt erkannt worden sind, doch gelegentlich können wir einen ganz falschen Kreis bekommen. Wenn wir also z.B. den Fall haben, dass wir 5 fast korrekte Kreise und einen ganz falschen Kreis erhalten, führt die Berechnung des arithmetischen Mittels zu keinem befriedigendem Ergebnis. Mit dem geometrischen Mittel bekommt man bessere, doch auch nicht ausreichende, Ergebnisse:

$$\widehat{x} = \sqrt[n]{\prod_{i=1}^{n} x_i}$$

Wir haben uns also für den sogenannten „Median" Algorithmus entschlossen, der viel bessere Ergebnisse liefert. Der lautet:

1. Sortiere die Elemente des Vektors nach der Größe
2 a. Wenn wir eine ungerade Anzahl von Elementen haben, ist der Medianwert das mittlere Element.
2 b. Wenn wir eine gerade Anzahl von Elementen haben, ist der Medianwert das arithmetische Mittel der zwei mittleren Elemente.

In diesem Fall (6 Elemente) müssen wir also Fall 2b betrachten. Für die Sortierung im Schritt 1 benutzen wir die schon implementierte Funktion *qsort*, die in der C-Standard Library *stdlib* enthalten ist (5). Die Vergleichsfunktion *cmp* mussten wir allerdings selbst implementieren (siehe Quellcode).

4.5.3 Ergebnisse

Unser zweiter, realisierter Ansatz hat in den meisten Fällen gut funktioniert. Bild 4.5.3.1 zeigt einige erfolgreiche Iris-Lokalisierungen.

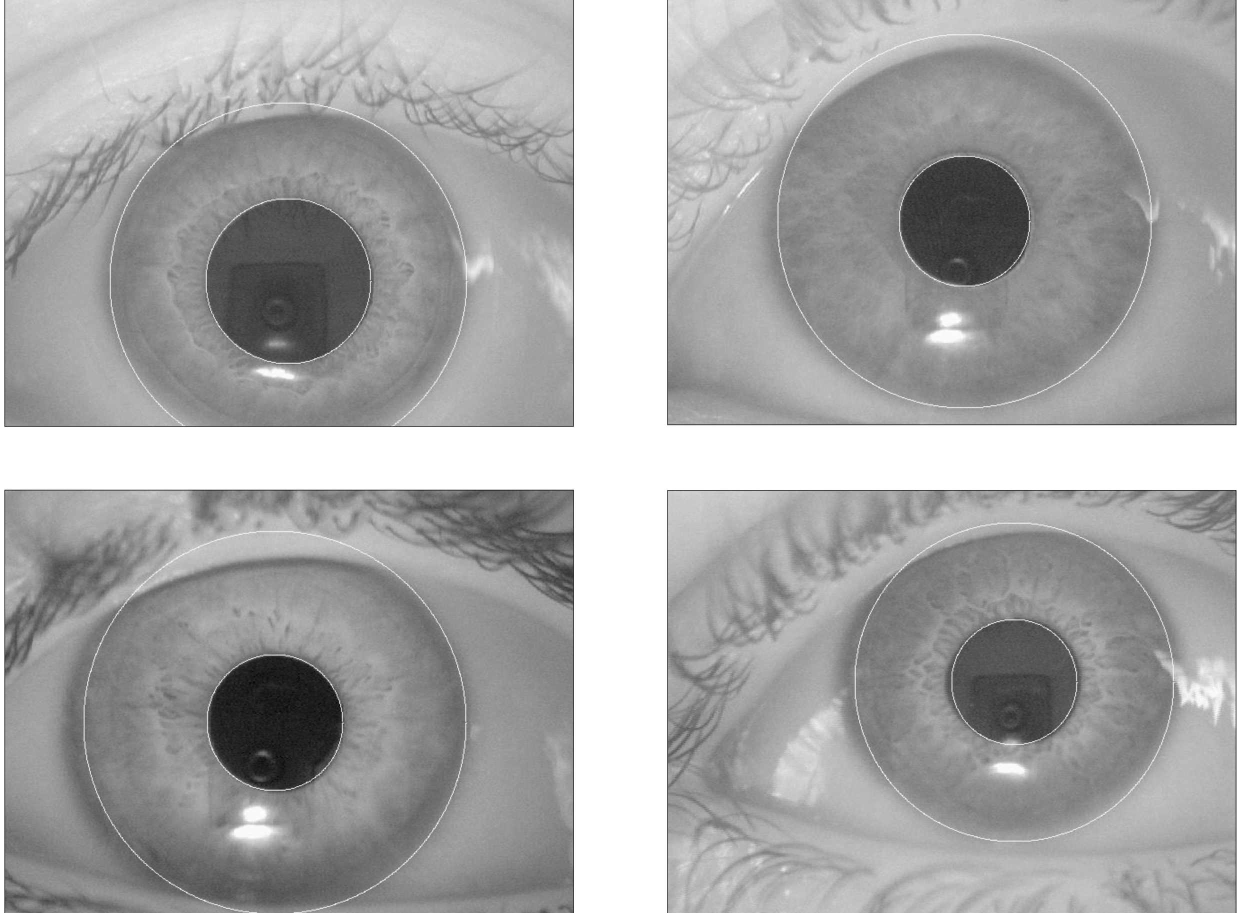


Bild 4.5.3.1 Ergebnisse der Iris-Lokalisierung

Wegen des starken Kontrasts zwischen Pupille und Iris, ist die Pupille leicht zu finden. Sie wurde immer richtig lokalisiert.
Schwieriger ist aber, eine korrekte Lokalisierung des Irisradius zu bekommen, da der Übergang zwischen Regenbogenhaut und der Rest des Auges viel weicher ist.
Außerdem haben wir beobachtet, dass kleine Änderungen in den Beleuchtungsverhältnisse einen großen Einfluss auf das Erkennungsverfahren haben können.
Diese Probleme führten in einigen Fällen zu fehlerhaften Erkennungen des Randes der Iris (Bild 4.5.3.2) :
Bezüglich des Zeitaufwands ist zu sagen, dass der Erkennungsalgorithmus (ohne die Anfangspunktbestimmung) auf dem DSP weniger als eine Sekunde gedauert hat.

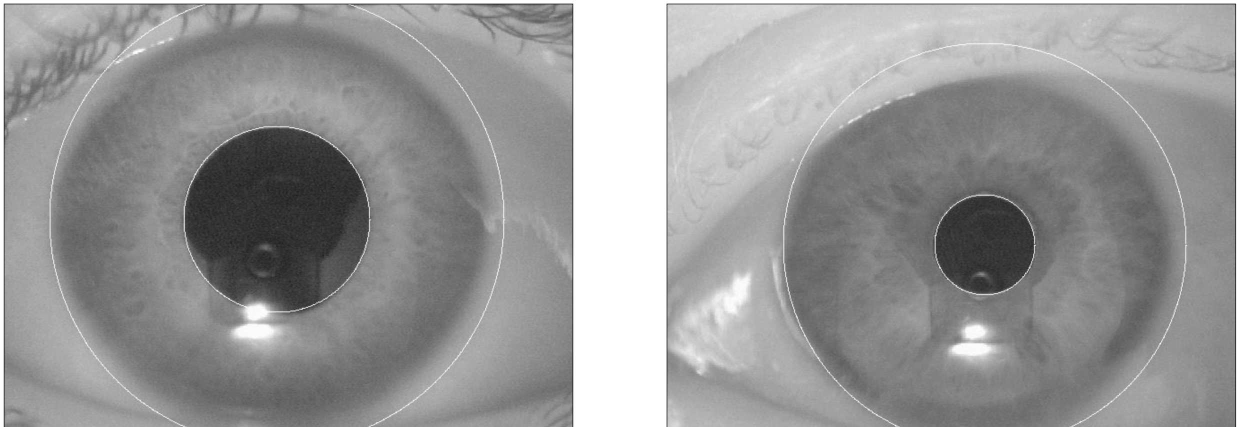


Bild 4.5.3.2 Fehlerhafte Ergebnisse der Iris-Lokalisierung

4.5.4 Frühere Ansätze

4.5.4.1 Kantenerkennung

Laut John Doughman's „How iris recognition works" (1) wird in professionellen Systemen die Iris über eine Kantenerkennung mit Gauß-Filter gesucht. Daher war dies auch unser erster Ansatz.
Die Kantenerkennung ist ein Verfahren durch das die Struktur eines Bildes hinsichtlich seiner Übergänge von einer Fläche zur anderen untersucht wird. Es stehen dazu unterschiedliche Filter zur Verfügung. Grundsätzlich wird dabei eine Matrize (das Filter) mit einem Bild (in unserem Fall ein Augenbild) gefaltet. Das Ergebnis liefert auf Grund der Struktur der Filter-Matrize eine neue Matrize bzw. ein neues Bild mit einer Hervorhebung der Kanten des Originalbildes. In der entsprechenden Literatur ist mehr darüber zu lesen (2).
Wir haben auf eine Filter-Auswahl, die in MATLAB implementiert ist, zurückgegriffen. Sobel-, Prewitt-, Roberts-, Gauß- und Canny – Filter wurden von uns auf Augenbilder zur Lokalisierung der Iris angewandt. Sobel-, Gauß-und Canny-Filter bewährten sich hinsichtlich ihrer Effizienz bei der Iris-Lokalisierung.
Den Filtern lassen sich noch Threshold-und auch Sigmawerte zuordnen, durch die sie sich steuern lassen. Hierin bestand auch ein Großteil unseres Experimentierens. Unser Bestreben war es möglichst einen Threshold-und einen Sigmawert zu finden, bei dem annähernd nur noch die Kanten der Iris und der Pupille im Bild enthalten sind.
Einige Filterungen mit Augenbildern (aus dem Internet) lieferten erste Ergebnisse. Das folgende Bild zeigt die Ergebnisse nach der Filterung:

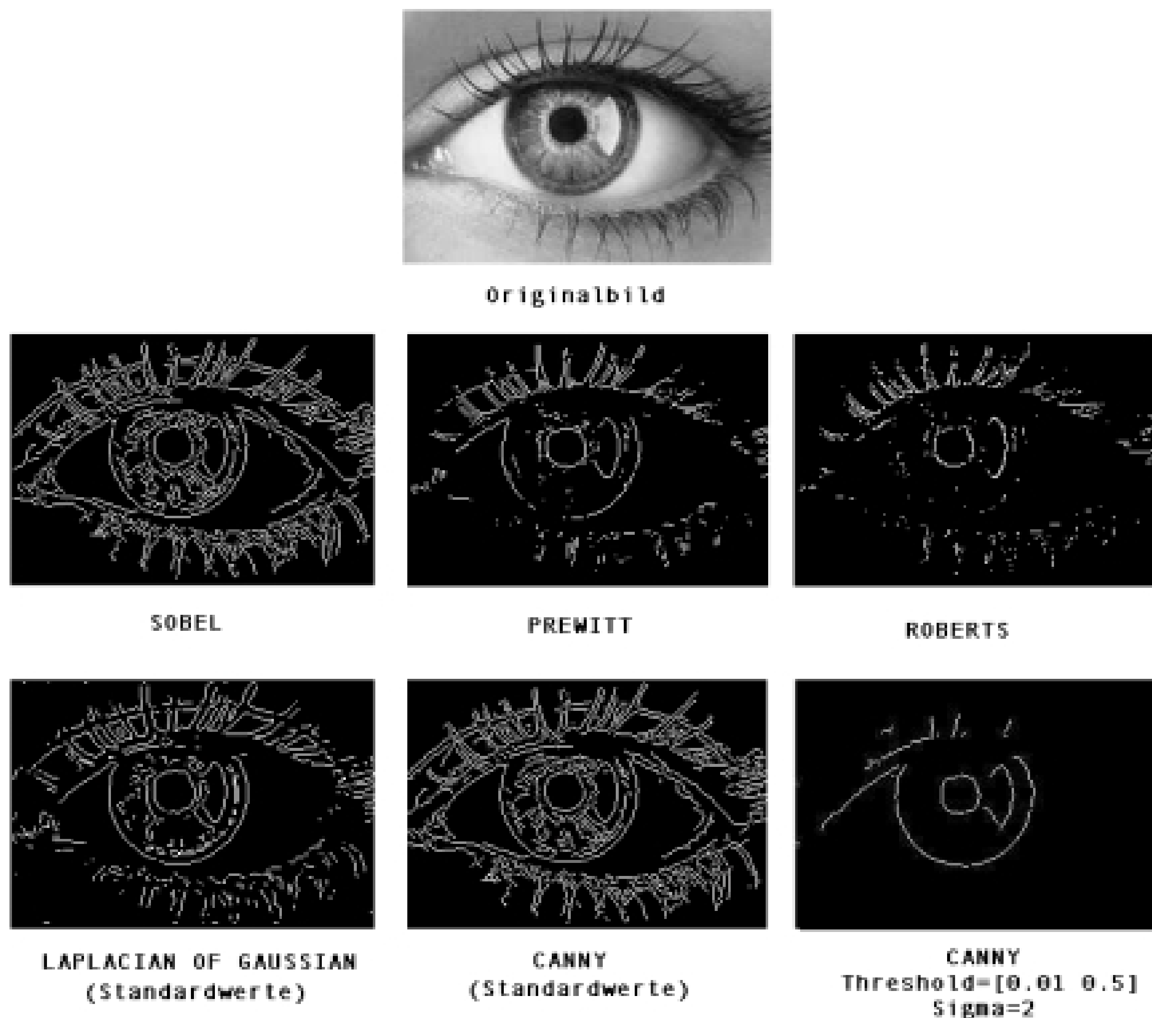


Bild 4.5.4.1 Kantenerkennung

Während wir die Filter einerseits auf ihre Effizienz hinsichtlich der Lokalisierung der Iris untersuchten, setzten wir uns auch mit dem Umfang und der Komplexität des Quellcodes des jeweiligen Filters auseinander. Sobel, Prewitt und Roberts sind relativ leicht zu implementieren, das Gauß-Filter hingegen schon schwieriger und das Canny-Filter sehr umfangreich und komplex.
Hier die Auswertung hinsichtlich Effizienz und Implementierung in einer Tabelle grob zusammengefasst:

	Sobel	**Prewitt**	**Roberts**	**Gauß**	**Canny**
Effizienz bei der Lokalisierung der Iris	+	-	-	+	+
einfach zu Implementieren	+	+	+	-	--
Gesamt	**++**	**o**	**o**	**o**	**-**

+ = positiv, - = negativ, o = neutral

4.5.4.2 Ring-Algorithmus

Nach dem Filtern des Bildes mit einem der Kantenerkennungsfilter haben wir ein Bild was nun eher einer Zeichnung gleicht und im Idealfall nur noch die Kanten der Iris und der Pupille enthält. Um jedoch die Fläche der Iris aus dem Bild zu schneiden, muss man den Kreis der Pupille sowie den der Iris mit einem weiteren Algorithmus genauer bestimmen.

Folgende Überlegung brachte uns zu unserem Ansatz:
Die Iris ist am wahrscheinlichsten in der Bildmitte zu finden. Außerdem handelt es sich bei Pupille und Iris annähernd um Kreise.

Wir definieren einen „Ring“ mit kleinem Radius, dessen Mittelpunkt zunächst die Bildmitte ist. Es werden die Punkte im Ring gezählt, die eine Kante kennzeichnen (in den Kantenbilder von „Bild 4.5.4.1 “ die weißen Punkte). Der Radius wird stetig erhöht und das Maximum an „Kantenerkennungspixeln“ wird gesucht. Der Ringmittelpunkt wird nach dem Durchlaufen von verschiedenen Radien neu gesetzt um auch eine Verschiebung der Iris vom Bildmittelpunkt zu berücksichtigen.

Bis das Maximum gefunden wird, müssen allerdings etliche Ringe definiert(Mittelpunkt, Radius) und auf ein Maximum untersucht werden. Die Prozedur dauert dem entsprechend lange.

Wenn wir eine geringe Spanne zwischen Minimum und Maximum des Pupillenradius sowie des Irisradius definieren, lässt sich der Algorithmus optimieren.
Grobe Sprünge bei den gewählten Radien am Anfang - die man erst später verfeinert - würden den Ablauf ebenfalls beschleunigen.
Setzt man den Algorithmus zunächst mit groben Verschiebungen des Ringmittelpunktes ein, und beginnt dann - nach der ersten Auswertung - in einem kleineren Bereich mit kleineren Mittelpunktverschiebungen zu arbeiten, so wäre der Algorithmus auch in dieser Hinsicht optimiert.

Der Ansatz wäre der jetzigen Anfangspunktbestimmung ähnlich. Wir waren allerdings der Überzeugung, dass es immer noch zu lange dauern wird, bis der genau Mittelpunkt und die Radien so gefunden werden.

Wir änderten unsere Herangehensweise und kamen zu dem schon beschriebenen Ansatz.

4.5.4.3 Frühere Anfangspunktbestimmungs-Ansätze

Bei der aktuellen Herangehensweise ist der Ansatz zur Anfangspunktbestimmung mehrfach überdacht worden.

Bei den ersten Varianten wendeten wir zunächst das Kontrastbildverfahren auf unsere Bilder an.
Nach dem Verfahren hatten wir im Idealfall eine schwarze Fläche an der Stelle der Iris bzw. der Pupille. Ansonsten war das Bild annähernd weiß.

Zur Bestimmung eines Anfangspunktes, wurde nach der größten zusammenhängenden schwarzen Fläche gesucht. Der Mittelpunkt dieser Fläche wurde als Anfangspunkt an den Drei-Punkte-Algorithmus weitergegeben, um dann die Kreise(Pupille, Iris) zu definieren.

Zwei Ansätze wurden von uns „ins Auge gefasst":

I) Das Bild wird in X-Richtung „gescannt" und auf die meisten schwarzen Pixel pro Bildspalte untersucht. Dann wird das Bild in Y-Richtung „gescannt" und auf ein Maximum an schwarze Pixel pro Bildzeile untersucht. X-und Y-Wert werden in einem Punkt zusammengefasst und an die nächste Funktion weitergegeben.

II) Das Bild wird „kästchenweise" auf Maxima an schwarzen Pixel untersucht. Von den „Maxima-Kästchen" wird das mit den meisten angrenzenden „Maxima-kästchen" als das „Beste" herausgegriffen und in dem Kästchen ein Punkt definiert, der dann an die folgende Funktion weitergegeben wird. Die Kästchengröße ist dabei so definiert, das etwa neun Kästchen in den jeweiligen Kreis passen.

Bei beiden Ansätzen wird auf die Kreisförmigkeit der Pupille bzw. der Iris entweder nur bedingt oder gar nicht eingegangen. Aus diesem Grunde wurde die Funktion von uns noch einmal überarbeitet. Die jetzige Anfangspunktbestimmung finden sie unter Abs. 4.5.2.2 .

4.6 Bildverarbeitung

4.6.1 Beschneiden des Bildes

Das Bild wird jetzt auf die relevanten Bildteile beschnitten.

Bild 4.6.1.1 Iris beschnitten

Im nächsten Schritt werden nun noch die unteren 90 Grad ausgeschnitten, da sich in diesem Bereich die Reflektionen der Beleuchtung und viel Feuchtigkeit befinden.

Bild 4.6.1.2 Iris ohne Reflexion

4.6.2 Polarkoordinaten-Transformation

Wir führen nun eine Polarkoordinaten-Transformation durch, um uns bei unseren weiteren Berechnungen die trigonometrischen Funktionen einzusparen und dadurch die Geschwindigkeit zu erhöhen.
Die transformierte Iris hat nun folgende Form:

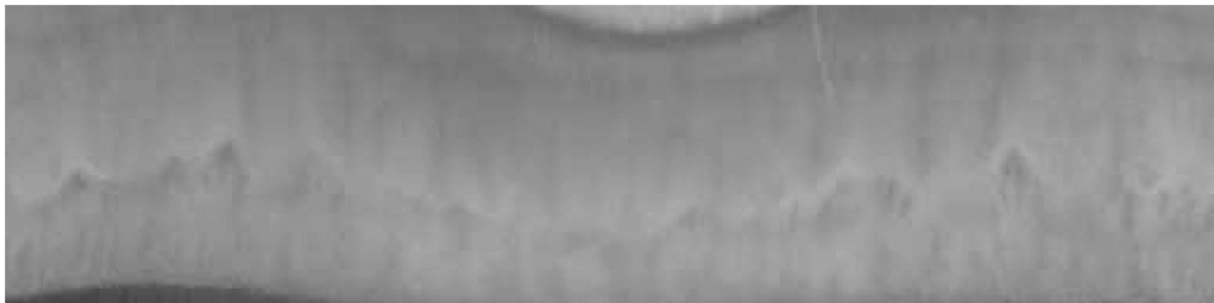


Bild 4.6.2 Iris in Polarform transformiert

Das Bild besitzt jetzt eine Große von 512 x 128 Pixel.

4.6.3 Gabor-Transformation

Die Gabor-Transformation dient dem Transformieren des Bildes in spektrale Informationen. Bei der Gabor-Transformation handelt es sich korrekterweise um eine Wavelet-Transformation mit Gaborfunktionen als Mutterwavelets.

Die Wavelet-Transformation ist eine Lineartransformation ähnlich der Fourier-Transformation mit dem wichtigen Unterschied, dass die Frequenzkomponenten im Zeit-/Ortsverlauf lokalisiert werden können, also ähnlich einer gefensterten Fourier-Transformation.

Der Vorteil der Gaborfunktionen besteht darin, dass sie sowohl im Orts- als auch im Frequenzbereich möglichst scharf lokalisieren.

Wir nutzen in unserem Fall eine zweidimensionale Gabortransformation in Polarkoordinatenform, da wir ein Bild haben und dieses auch schon in Polarkoordinaten überführt haben. Folgende Abbildungen zeigten den Realteil eines Beispielwavelets jeweils in kartesischen- und in Polarkoordinaten:

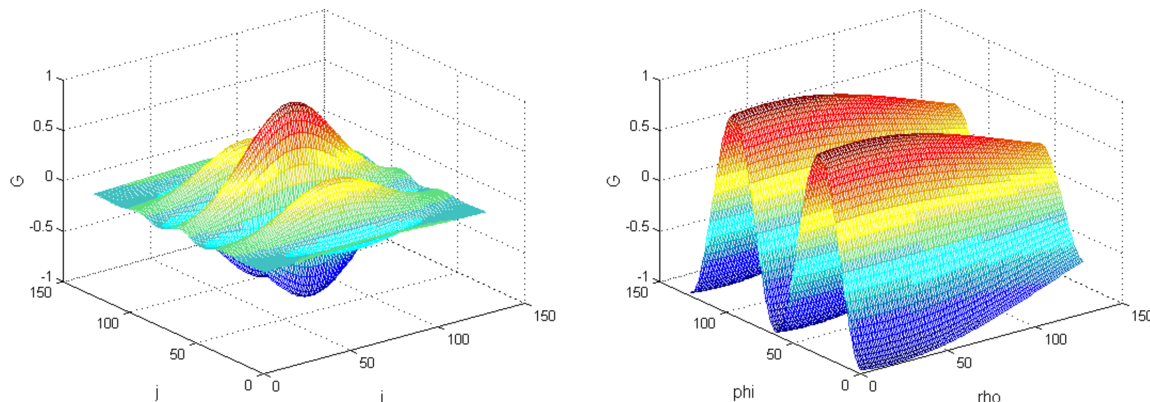


Bild 4.6.3 Gabortransformation in Kartesische- und Polarkoordinaten

4.6.4 Wavelets

Aus Geschwindigkeitsgründen berechen wir uns einige Tabellen mit Gaborwavelets im Voraus. Wir berechen Wavelets für zwei Frequenzen und jeweils 4 Phasenverschiebungen. Diese Tabellen benutzen wir später für die Korrelation mit dem Bild. Unsere Wavelets haben eine Große von 16x16 Punkten.

4.6.5 Korrelation

Es wird jeweils ein Teil des Bildes in der Größe der vorberechneten Tabellen berechnet. In der Art von Kacheln wird so das ganze Bild nach und nach Transformiert.

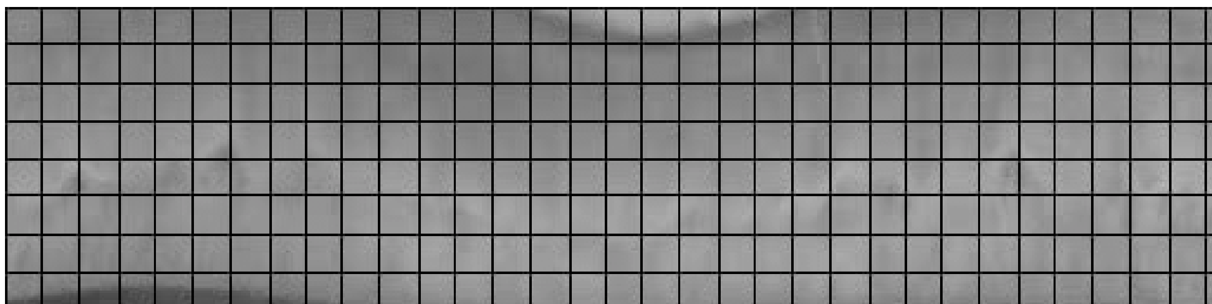


Bild 4.6.5 Iris Kacheln-Korrelation

Jede dieser Kacheln wird als erstes Mittelwertfrei gemacht und danach nacheinander mit allen vorgerechneten Wavelets korreliert. Es entsteht dabei jeweils ein komplexer Wert, der den Korrelationsgrad der Kachel mit dem entsprechenden Gaborwavelet angibt. Es entstehen somit „Anzahl der Kacheln“ * „Anzahl der Wavelets“ komplexe Ausgangswerte pro Bild, in unserem Fall sind es also 256*8 = 2048.

4.6.6 Entscheider

Es wird von uns nur die Phaseninformation weiterverwendet, da die Amplitude stark Beleuchtungsabhängig ist. Wir unterscheiden nur 4 Quadranten, d.h. Wir bekommen aus unseren 2048 Koeffizienten noch 4096 Bit an Informationen.

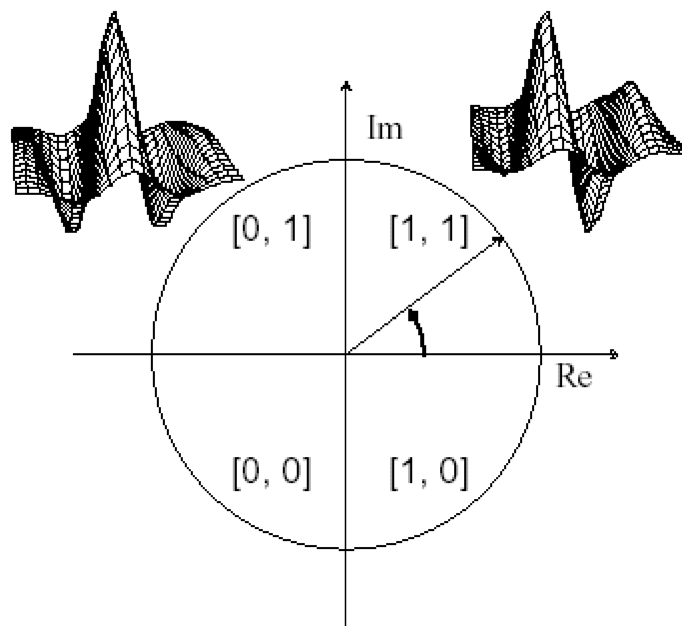


Bild 4.6.6 4 Phaseninformation

Diese Bits hintereinander gesetzt ergeben nun unseren Iris-Code.

4.7 Programmstruktur der Bildverarbeitung (*GetScanCode.pjt*)

4.7.1 Einleitung

Damit eine möglichst reibungslose Anpassung der Software an Veränderungen einzelner Algorithmen vorgenommen werden kann, ist ein entsprechend flexibler Aufbau nötig. Dieser wird vor allem durch Zusammenfassung von Funktionsaufrufen und deren Abkapselung von thematisch andersartigen Gebieten erreicht.

Dabei beziehen sich die folgenden Beschreibungen nur auf die Board-Software, da eine funktionierende Implementierung der Host-Applikation nicht zustande kam. Auf die entsprechenden Abschnitte zu RTDX wird an dieser Stelle ebenfalls nicht

eingegangen. Die Aufrufe sind im hier zu sehenden Quelltext entfernt worden, in der Softwaresammlung aber noch vorhanden, bzw. nur auskommentiert worden.

Der Aufbau wird hier nur sehr prinzipiell beschrieben, da Datenformate und spezielle Aufrufkonventionen aus den Quelltexten ersichtlich sein sollten. Einzelne Algorithmen werden hier ebenfalls nicht besprochen, sondern nur der generelle Ablauf der Funktionen ***main()*** und ***GetScanCode()***.

4.7.2 Hauptprogramm (*MAIN.c*)

In der Hauptfunktion laufen sämtliche Prozesse ab, die zur Bestimmung des Iris-Codes einer Person notwendig sind. Dieser Code wird im Programmablauf vollständig berechnet, auch wenn zum jetzigen Entwicklungsstand nur hypothetische Aussagen über dessen Signifikanz gemacht werden können.

Die statistische Auswertung dieser Daten, sowie die Steuerung der Wiederholungen sollte in der Host-Applikation stattfinden. Somit erscheinen auch noch eine Reihe von Debug-Informationen im Quelltext, die dann die Beobachtung der einzelnen Aufrufe im Code Composer ermöglichen. Auch das Timing der Bildaufnahme hätte im Idealfall von der Host-Applikation übernommen werden sollen.

4.7.3 Speicherbereiche

Für den gesamten Programmablauf wird für den Iris-Code einmal Speicher reserviert. Gleiches gilt für die Tabellen zur Gaborfilterung, das Augenbild und dessen verkleinerte Version.

Iris-Code	***arrayPointer***	= 512 Byte
Gabor-Tabellen (real)	***regabtab***	= 16 kByte
Gabor-Tabellen (imaginär)	***imgabtab***	= 16 kByte
Bild (640*480)	***buffer***	= 300 kByte
Verkleinertes Bild (160*120)	***scaled_buffer***	≈ 19 kByte

Dazu kommt noch eine Variable zur Überprüfung der Bildschärfe (***sharp***), deren Threshold-Wert (***sharpness_threshold***) einstellbar ist.

4.7.4 Ablauf

Zuerst wird das Videoboard initialisiert - ***StartVideo()***.

Dann werden die Gabor-Filtermatrizen als Tabellen initialisiert - ***init_gabtab()***. Sie bleiben über den ganzen Programmablauf unverändert und werden deshalb nur einmal berechnet

Die folgenden Abläufe sollten dann eigentlich in einer Schleife ablaufen, um verschiedene Personen untersuchen zu können. Dies ist jedoch in dieser Version

nicht der Fall, da die entsprechend notwendige Steuerung durch die Target-Applikation nicht funktioniert.

Aus Geschwindigkeitsgründen wird vor der eigentlichen Code-Gewinnung eine Schärfebestimmung durchgeführt. Da die Berechnung des Codes sehr lange dauert, wollten wir es uns nicht leisten, diese Berechnung auf unscharfen und damit unbrauchbaren Bildern durchzuführen.

In einer Schleife wird solange ein Bild aufgenommen, bis dieses unsere Schärfekriterien erfüllt. Weitere Überprüfungen von Bildeigenschaften finden hier nicht statt.
Dazu wird zuerst ein Bild aus dem Videoboard in den Speicher kopiert - ***GetImage()***.
Dann wird davon eine verkleinerte Version erzeugt - ***scale()***.
Von diesem verkleinerte Bild wird dann die Schärfe bestimmt - ***sharpness()***.
Liegt der Schärfewert unter dem Schwellwert – ***sharpness_threshold***, wird diese Prozedur wiederholt.

Ist nun ein scharfes Bild aufgenommen worden wird dieses mit weiteren Parametern an die Funktion ***GetScanCode()*** übergeben. Diese weiteren Parameter sind das verkleinerte Augenbild, die Gabor-Tabellen und der Iris-Code. Der komplette Ablauf zur Iris-Code-Gewinnung ist in dieser Funktion zusammengefasst und wird später noch erläutert. Als Ergebnis des Aufrufs wird der berechnete Code in die entsprechende Variable geschrieben.

Dieser Code sollte nun, so war der Plan, zur Weiterverarbeitung an die Host-Applikation gesendet werden. Wäre diese Verarbeitung abgeschlossen, so hätte eine erneute Bildaufnahme und Code-Gewinnung stattfinden können.

Zum Abschluss des Programms wird das Videoboard mit ***StopVideo()*** heruntergefahren und die Speicher werden freigegeben.

4.7.5 Iris-Code-Gewinnung (*GetScanCode.c*)

Um den Ablauf des Hauptprogramms übersichtlicher zu gestalten, wird die gesamte Code-Gewinnung in einem Funktionsaufruf zusammengefasst. Somit bleiben die entsprechenden internen Abläufe auch von dem Problem der Target-Host-Kommunikation abgekapselt, was ein einfacheres Arbeiten für beide Seiten ermöglichte.

4.7.5.1 Speicherbereiche

An die Funktion werden die Speicherbereiche für Iris-Code, Augenbild, verkleinertes Augenbild und für die Gabor-Tabellen.

Hinzu kommen noch das polarkoordinatentransformierte Iris-Bild, Real- und Imaginärteil der gefilterten Iris. Sie existieren nur lokal in dieser Funktion und werden beim Verlassen freigegeben.

Iris-Bild (Polarkoordinaten)	***pol_bufferr***	= 64 kByte
Gabor-gefilterte Iris (real)	***re_buffer***	= 8 kByte
Gabor-gefilterte Iris (imaginär)	***im_buffer***	= 8 kByte

Als Variablen existieren dann noch die Abmessungen und die Position der Iris (***r_pup***, ***r_iris***, ***mx_pup***, ***my_pup***).

4.7.5.2 Ablauf

Zuerst wird ein Punkt in der Pupille lokalisiert. Dieser Teil der Suche findet auf dem verkleinerten Augenbild statt – ***locate_pupil()***.

Die Startkoordinaten liegen nun vor, und die exakten Werte für Position und Abmessungen der Iris (Originalbild) können bestimmt werden – ***iscan_main()***.

Mit den ermittelten Irisdimensionen wird nun eine Polarkoordinatentransformation des entsprechenden ringförmigen Bildbereichs durchgeführt. Als Ergebnis erhalten wir das rechteckiges Abbild eines 270º-Ausschnittes der Iris - ***poltrans()***.

Auf diesem Abbild wird dann eine Gabor-Filterung vorgenommen. Bei dieser nichtüberlappenden Transformation kommen nun auch die Gabor-Tabellen zum Einsatz – ***filter()***.

Aus den entstehenden Real- und Imaginärteilbildern wird jetzt durch einfache 2bit-Phasen-Quantisierung der Iris-Code berechnet und in den entsprechenden Puffer geschrieben– ***icode()***.

Nachdem nun der Code vorliegt, wird die Funktion verlassen.

4.7.6 Anmerkungen

Die Board-Software arbeitet im Rahmen kleinerer Genauigkeitsschwankungen sehr zuverlässig. Die Präzision einzelner Algorithmen wird dann an entsprechender Stelle besprochen. Speicherprobleme, Ablauf- und Zugriffsfehler kommen dank recht sorgfältiger Programmierung nicht vor. Einige Initialisierungsschwierigkeiten beim Video-Zugriff liegen ursächlich wahrscheinlich im dortigen Hardware-Bereich und konnten von uns nicht lokalisiert werden.

Der Gesamtspeicherbedarf für Datenpuffer lässt sich in etwa auf 0,8 MByte abschätzen. ***iscan_main()*** beispielsweise benutzt noch ein zweites komplettes Augenbild. Unser Linker-Command-File (***GetScanCode_lnk.cmd***) musste dann entsprechend unseren Anforderungen modifiziert werden. Eine große Heap-Size von 1MByte schien uns angemessen.

Wichtig für einige Funktionsaufrufe ist die Einbindung folgender Bibliotheken in das Projekt:

img62x.lib

rtdx.lib
rts6201.lib

Mit einer Rechenzeit von etwa 1 Minute dauert die Code-Berechnung sehr lange. Die Bildschärfebestimmung kann demgegenüber als Zeitfaktor vernachlässigt werden (etwa 1 Sekunde inkl. stets notwendiger Bildskalierung). Ursachen hierfür sind in den jeweiligen Algorithmen und deren Implementierung zu suchen. Genaueres hierzu gibt es in den entsprechenden Abschnitten. Optimierungsmöglichkeiten bieten die hier erläuterten Abläufe kaum, da jeweils nur die entsprechenden Funktionen aufgerufen werden.

5. PC-Schnittstelle

5.1 Einleitung

Die Möglichkeit eine Person durch die Aufnahme der Iris zu identifizieren setzt voraus, dass bereits eine Datenbank mit unterschiedlichen Informationen (Name, Code, ...) vorhanden ist. Da bei einer größeren Menge von Personen ein entsprechender Speicherbedarf vorausgesetzt wird, wurde von uns eine Speicherung der Daten auf dem PC vorgesehen. Weil die Iris Codes auf dem DSP berechnet werden, ist eine PC-Anbindung vorausgesetzt. Hierzu entschieden wir uns zur Benutzung der parallelen Schnittstelle auf dem Entwicklungsboard.

Die Kommunikation zwischen dem PC und der DSP-Karte wird vereinfacht, da bereits Texas Instruments eine Echtzeit Datenübertragung durch das so genannte RTDX (Real-Time Data eXchange) zur Verfügung stellt.

Zur Realisierung dieser Datenübertragung wird auf dem PC eine Hostapplikation und auf der DSP-Karte eine Targetapplikation vorliegen. Diese Applikationen werden über die erwähnte RTDX-Schnittstelle kommunizieren. Hardwaremäßig wird das DSP-Board an den PC über die parallele Schnittstelle angebunden. Die Daten werden vom DSP-Board über das JTAG-Interface[1)] empfangen bzw. gesendet.

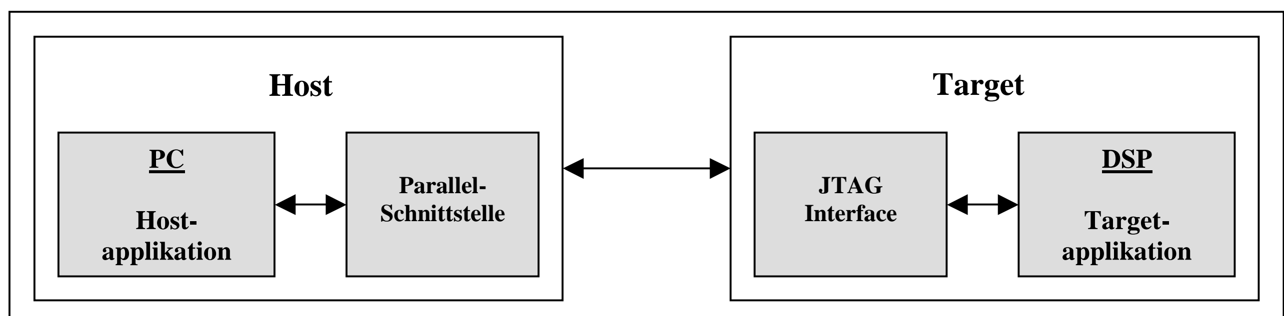


Bild 5.1 Kommunikationsübersicht

5.2 RTDX

5.2.1 Einführung in RTDX

Von Texas Instruments werden neben der Entwicklungsumgebung die entsprechenden RTDX-Bibliotheken sowohl für die Host- als auch für die Targetapplikation mitgeliefert.
Diese Bibliotheken beinhalten vordefinierte Funktionen und Makros, wodurch Kommunikation auf- bzw. abgebaut wird.
Das Linken einer kleinen RTDX Software Bibliothek in die Targetapplikation ermöglicht das Senden und Empfangen. Beim Transfer werden aus dieser Bibliothek Funktionen aufgerufen.

Hostseitig wird die RTDX Dynamic Link Library (rtdxint.dll) angebunden. Diese Bibliothek bietet die Kommunikation mit dem Code Composer an. Die RTDX DLL ermöglicht das Senden und das Empfangen von Daten, auch über einen COM Client (z.B. Visual C++, Visual Basic).

Die folgende Abbildung gibt eine Übersicht über die RTDX Architektur.

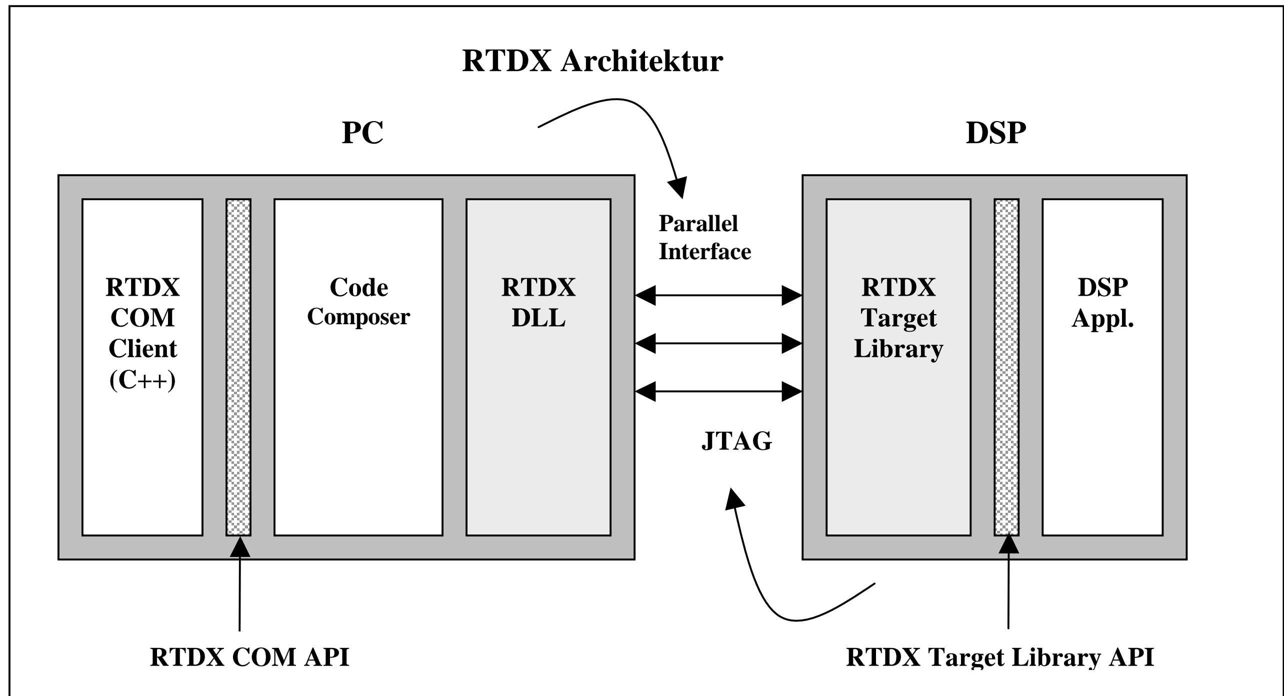


Bild 5.2 RTDX Architektur

5.2.2 RTDX Target Library API

Die Targetapplikation kommuniziert mit RTDX über die RTDX Target Library API. Diese API ermöglicht das Definieren von virtuellen Kanälen, über die Daten gesendet bzw. empfangen werden können. Hierzu werden Funktionen zum aktivieren und deaktivieren dieser Kanäle angeboten. Die von uns verwendete Lese-Funktion blockiert die Targetapplikation solange, bis die Hostanwendung mindestens ein Datum sendet.
Es folgt eine Übersicht der verwendeten Funktionen:

- RTDX_CreateOutputChannel(name) // Ausgangskanal deklarieren
- RTDX_CreateInputChannel(name) // Eingangskanal deklarieren
- RTDX_enableOutput(&name) // Ausgangskanal aktivieren
- RTDX_enableInput(&name) // Eingangskanal aktivieren
- RTDX_disableOutput(&name) // Ausgangskanal deaktivieren
- RTDX_disableInput(&name) // Eingangskanal deaktivieren
- RTDX_write(&output_channel, &buffer, sizeof(buffer)) // Sendet den Inhalt von *buffer* über den Ausgangskanal an die Hostanwendung

- RTDX_read(&input_channel, &buffer, sizeof(buffer)) // Empfängt über den Eingangskanal und schreibt in *buffer* ein

5.2.3 RTDX COM API

Auf dem PC kommuniziert ein Client mit RTDX über die RTDX COM API. Diese Schnittstelle bietet Funktionen zum Öffnen, Schließen, Aktivieren, Deaktivieren von Kanälen. Diese Kanäle entsprechenden den virtuellen Kanälen aus der Targetapplikation. Außerdem existieren Funktionen zum Lesen und Schreiben von Daten.
Es folgen die von uns verwendeten Funktionen:

- Open("name", "Modus") // Öffnet einen Kanal zum Lesen (R) oder zum Schreiben (W)
- WriteI2(toSend, &bufferstate) // Sende eine 2 Byte Integer an den Target
- ReadI2(toReceive) // Empfangen eines 2 Byte Integers
- ReadSAI1(&sa) // Empfangen eines Arrays der Elementgröße von 1 Byte
- Close() // Schließen eines Kanals

5.2.4 Übertragung: Target → Host

Um eine Übertragung von der Targetapplikation zum PC durchzuführen wird erst beidseitig ein virtueller Kanal definiert, und zwar ein Ausgangskanal (Daten werden vom DSP übertragen). In der Targetapplikation wird dieser Kanal aktiviert. Hostseitig wird es im Lesemodus geöffnet. Mit dem Schreibbefehl *RTDX_write* werden die Daten an die Hostapplikation gesendet. Vor dem Senden werden die Daten in einem internen Target Puffer zwischengespeichert. Die Daten werden dann nach dem FIFO Prinzip übertragen. Hostseitig werden die Daten von der RTDX DLL empfangen und nach Auswahl in eine Log-Datei oder in einen Puffer geschrieben. Diese Aktionen werden in Echtzeit durchgeführt.
Der COM Client erhält die Daten nun von der RTDX DLL über eine Lesefunktion aus der RTDX COM API.

5.2.5 Übertragung: Host → Target

Auch bei dieser Übertragungsrichtung wird zunächst beidseitig wieder ein Kanal deklariert. Hierbei handelt es sich jedoch um einen Eingangskanal. Targetseitig wird ein RTDX_read aufgerufen. Ein „read request" wird in den Target Puffer geschrieben. Hostseitig wird diese Anforderung von der RTDX DLL empfangen. Mit einem Schreibbefehl werden dann vom COM Clienten die Daten an die RTDX DLL übergeben. Liegen die Daten an, so werden diese an die Targetapplikation gesendet. Da die Zeit für die Bereitstellung der Daten nicht bekannt ist, kann die Übertragung in dieser Richtung nicht ganz in Echtzeit durchgeführt werden.

5.3 Targetapplikation

5.3.1 Übersicht

Die Targetapplikation auf dem DSP ist in der Programmiersprache C geschrieben. Zunächst wird hier eine Übersicht über den Ablauf des Programms gegeben. Es folgen dann Funktionsbeschreibungen und Definitionen.

Die Übertragung erfolgt in der Reihenfolge, dass zunächst die Kommunikation zwischen dem Target und dem Host durch das Senden eines Requests und Empfangen eines Acknowledges überprüft wird. Nach Aufforderung wird der Iris-Code an die Hostapplikation gesendet.

Der Iris-Code wird in einem unsigned char Array der Größe von 512 Byte abgespeichert.

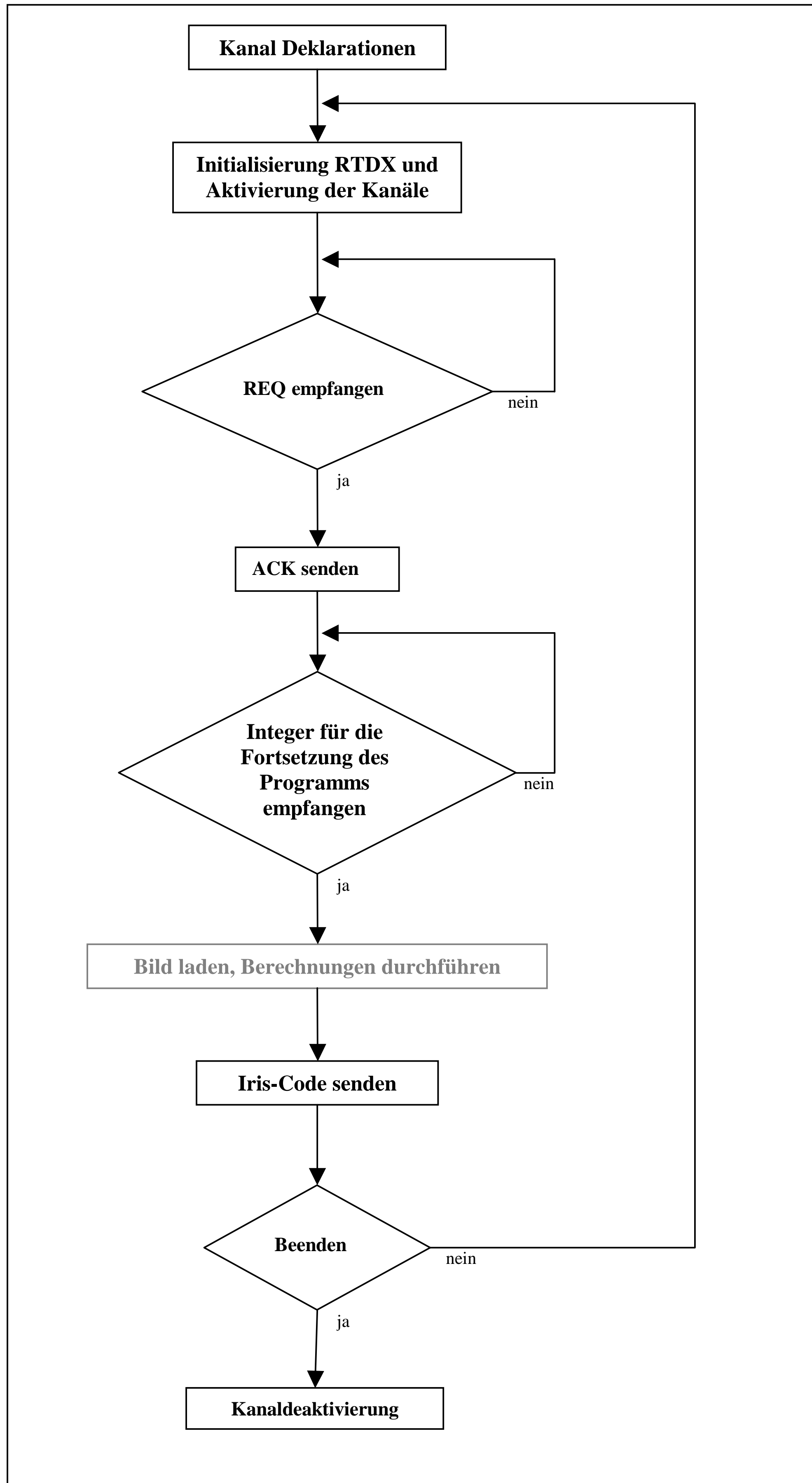


Bild 5.3 Übersicht Targetapplikation

5.3.2 Funktionsdefinitionen und –Beschreibungen

- Zur Initialisierung von RTDX und Kanalaktivierung existiert die folgende Funktion:

```
void init_RTDX()
{
   TARGET_INITIALIZE();
   RTDX_enableInput( pichan );
   RTDX_enableOutput( pochan );
   RTDX_enableOutput( pochan_SAFEARRAY );
   printf("Kanäle wurden aktiviert ... \n");
}
```

- Die Kanaldeaktivierung erfolgt über die Funktion *deinit_RTDX()*:

```
void deinit_RTDX()
{
   RTDX_disableInput( pichan );
   RTDX_disableOutput( pochan );
   RTDX_disableOutput( pochan_SAFEARRAY );
   printf("Kanäle wurden deaktiviert ... \n");
}
```

- Die Daten an den PC werden Mithilfe der Funktion *int_senden()* gesendet:

```
void int_senden(short toSend)
{
   int status;

   status = RTDX_write( pochan, &toSend, sizeof(toSend) );
   if (toSend == 10)
        printf("ACK gesendet ...\n");

   if ( status == 0 )
   {
        puts( "ERROR: RTDX_write failed!\n" );
        exit( -1 );
   }
   while ( RTDX_writing != NULL )
   {
#if RTDX_POLLING_IMPLEMENTATION
        RTDX_Poll();
#endif
   }
}
```

An die Funktion wird der Wert übergeben, der an die Hostapplikation gesendet werden soll.

- Die Requests vom PC werden über eine entsprechende Funktion *int_empfangen()* empfangen:

```
short int_empfangen()
{
   int status;
   short toReceive;

   status = RTDX_read( pichan , &toReceive , sizeof(toReceive) );
   if ( status == 0 )
   {
        puts( "ERROR: RTDX_read failed!\n" );
        exit( -1 );
   }
   if (toReceive == 5)
        printf("REQ empfangen ...\n");
   else if (toReceive == 10)
        printf("Iris wird aufgenommen ...\n");

   return toReceive;
}
```

- Zum Senden der Iris-Codes wird eine separate Funktion *array_senden()* benutzt:

```
void array_senden(unsigned char *array)
{
   int status;

   status = RTDX_write( pochan_SAFEARRAY , array , DATAVOLUME);

   if ( status == 0 )
   {
      puts( "ERROR: RTDX_write failed!\n" );
      exit( -1 );
   }
   puts("Iris Code gesendet ... \n" );
}
```

Die Funktion erfordert die Übergabe der Adresse des Iris-Codes, welches bereits als ein Array vorhanden ist.

5.3.3 Programmablauf

Zur Anfang des Programms werden 3 virtuelle Kanäle deklariert. Diese erfolgen über die unter RTDX beschriebenen Funktionen:

```
RTDX_CreateOutputChannel( ochan );
RTDX_CreateInputChannel( ichan );
RTDX_CreateOutputChannel( ochan_SAFEARRAY );
```

Anschließend werden Pointer auf die Adressen der Kanäle definiert:

```
RTDX_output_channel *pochan;
RTDX_input_channel *pichan;
RTDX_output_channel *pochan_SAFEARRAY;
```

Über ichan werden die Requests der Hostapplikation empfangen, über ochan die Antworten gesendet und über ochan_SAFEARRAY wird der Iris-Code an die Hostapplikation gesendet.

Nach der Initialisierung des RTDX und Aktivierung der Kanäle wird die Targetapplikation solange gestoppt, bis ein Request von der Hostapplikation da ist. Liegt ein Request an, so wird eine vordefinierte Ganze Zahl an die Hostapplikation gesendet. Hiermit lässt sich die Kommunikation zwischen Target und Host überprüfen. Bei einem erneuten Request wird Targetseitig ein Bild aufgenommen. Es folgen entsprechende Berechnungen.
Nach Abschluss der Algorithmen wird der Iris-Code an die Hostapplikation gesendet.

Empfang:

Der Empfang eines Request von der Hostapplikation erfolgt über den folgenden Funktionsaufruf:

```
Command = int_empfangen();
```

Innerhalb dieser Funktion findet ein Aufruf
status = RTDX_read(pichan , &toReceive , sizeof(toReceive)); statt.
Über *pichan* wird in *toReceive* der Größe von *toReceive* ein Request empfangen.
Diese wird der Variable *Command* übergeben.
Die Funktion RTDX_read übergibt als Statuswert folgende Informationen wieder:

<u>Rückgabewerte:</u>

> 0	Die Datenmenge wurde empfangen.
= 0	Fehler, Puffer ist voll
RTDX_READ_ERROR	Fehler, Kanal belegt oder nicht aktiv.

Senden:

Das Senden von Ganzen Zahlen an die Hostapplikation erfolgt über die Funktion int_senden(toSend). Innerhalb der Funktion erfolgt folgender Aufruf zum Senden einer Zahl:
status = RTDX_write(pochan, &toSend, sizeof(toSend));
Als Rückgabewert wird hier ebenfalls eine Null zurückgegeben, falls Fehler entstehen.

Das Senden des Arrays erfolgt ebenso über diese Funktion. Jedoch werden hier als Größe 512 Byte angegeben (Größe des Arrays).

Die Targetapplikation endet damit, dass alle aktivierten Kanäle wieder deaktiviert werden.

5.3.4 Probleme

Die Targetapplikation sollte sich nach dem starten in einer Wiederholungsschleife befinden, bis von der Hostapplikation das Request zum Beenden des Programms empfangen wird. Jedoch ergab sich hier das Problem, dass das Iris-Code nur nach Beendigung des Programms an die Hostapplikation geschrieben wird. Eine entsprechende Flush Funktion für den Kanal ist leider Targetseitig von Texas Instruments nicht vorgesehen.
Daher wird nach jeder Aufnahme und Übertragung des Iris-Codes die Targetapplikation beendet.

5.4 Hostapplikation

5.4.1 Übersicht

An die Hostapplikation wird die Aufgabe gestellt, die Datenverwaltung und die Steuerung zu übernehmen. Es sendet einen Request an die Targetapplikation um die Kommunikation zu überprüfen. Nach einem Acknowledge Empfang wird erneut ein Request gesendet, falls eine Bildaufnahme stattfinden soll.
Die empfangenen Iris Codes werden in einer Datei mit dem zugehörigen Namen und Vornamen abgelegt.

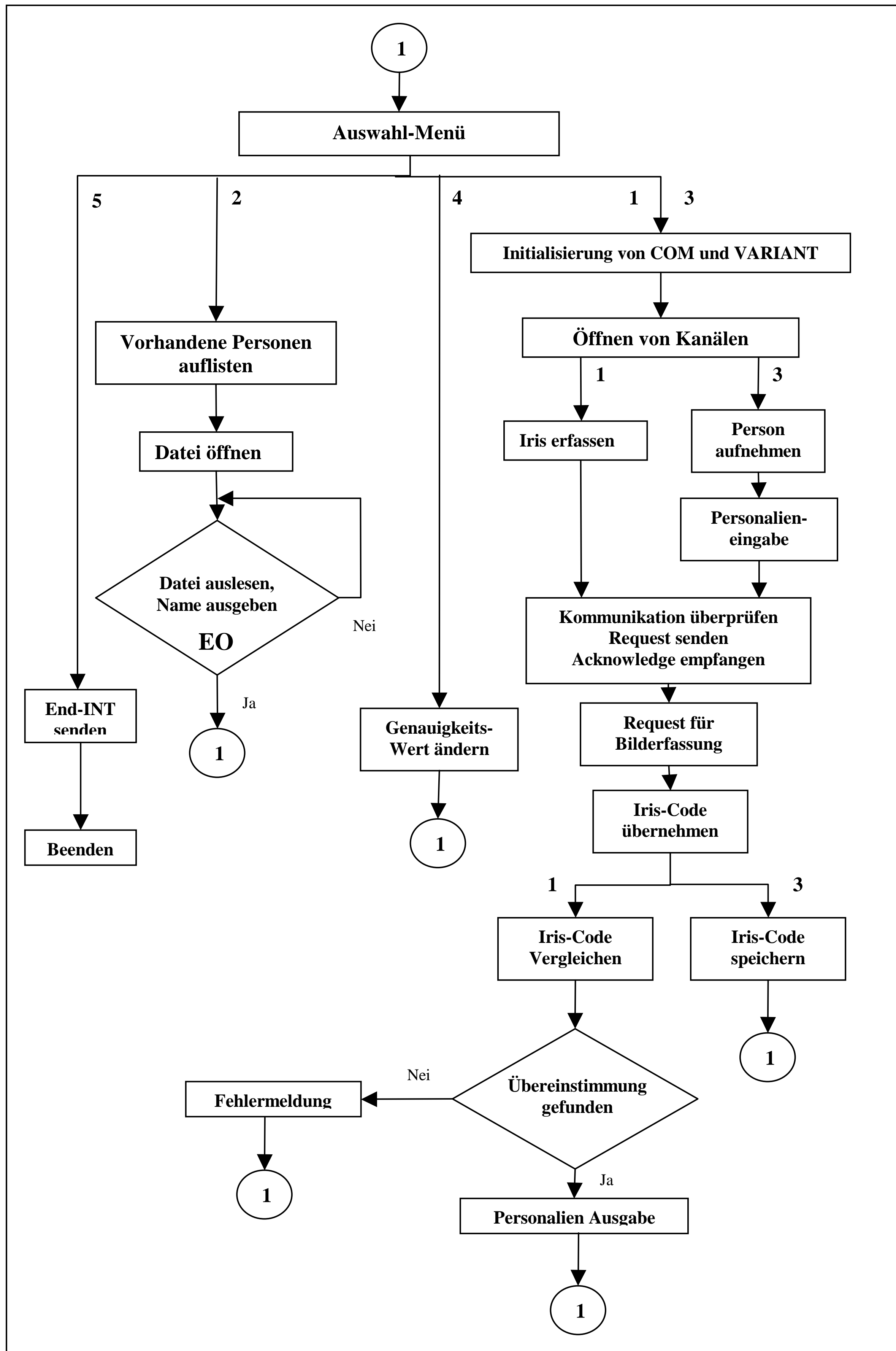


Bild 5.4 Übersicht Hostapplikation

5.4.2 Beschreibung der einzelnen Pfade

Werden im Auswahl-Menü die Pfade 1 oder 3 ausgewählt, so wird COM und VARIANT initialisiert. VARIANT dient zum empfangen des Iris-Codes. Außerdem werden virtuelle Kanäle geöffnet. Bei der Auswahl „Iris erfassen“ erfolgt zunächst die Überprüfung der Kommunikation zwischen Target und Host. Hierzu wird ein Request gesendet und eine Acknowledge abgewartet. Anschließend wird bei Erfolg ein Request gesendet, damit Targetseitig die Iris aufgenommen wird. Abschließend wird von der Hostapplikation der berechnete Iris Code übernommen. Dieser Code wird mit denen aus der Datenbank verglichen und eine entsprechende Ausgabe gemacht.

Wird im Auswahl Menü die Entscheidung „Neu Person eintragen“ gewählt, so werden vor der Überprüfung der Kommunikation die Personalien aufgenommen. Ein Vergleich der Daten findet hier nicht statt. Die empfangenen Daten werden zusammen mit den aufgenommenen Personalien strukturiert in einer Datei auf dem PC gespeichert.

Über das Menü 4 lässt sich der Genauigkeitswert ändern, der bei dem Vergleich verwendet wird.

Menüpunkt 2 listet alle Personen aus der Datenbank auf. Mit Auswahl 5 wird die Hostapplikation beendet. Vor dem Beenden wird an die Targetapplikation ein Request zum Beenden gesendet.

5.4.3 Benutzte Funktionen

Es wurden mehrer Funktionen geschrieben, um z.B. den Genauigkeitswert zu ändern, eine ganze Zahl zu senden oder zu empfangen, eine neue Person in die Datenbank aufzunehmen etc. Diese Funktionen können aus dem Quellcode im Anhang entnommen werden.

5.4.4 Vergleich

Der Vergleich von zwei Iris Codes erfolgt über die Funktion:

```
// Funktion, um Iris Codes zu vergleichen
long CompIriscode(unsigned char *bank, unsigned char *trans , int
FehlerGrenze)
{
      const unsigned char konst=1;
      int i,j;
      int count = 0;
      long Ergebnis[Datavolume];
      long Shifted;
      long LogBuffer;

      for ( j=0 ; j < Datavolume; j++)
      {
            Ergebnis[j] = bank[j] ^ trans[j];      //   ^ - EXOR
            for ( i=0 ; i < (sizeof(unsigned char)*8) ; i++ )
```

```
            {
                  Shifted = Ergebnis[j] >> i;
                  LogBuffer = konst & Shifted;
                  if ( LogBuffer != 0 )
                        count++;
                  if ( count  > FehlerGrenze )
                        return Failure;
            }
      }
      return count;
}
```

Der Funktion werden die zu vergleichenden Codes und die maximale Fehlergrenze übergeben.

Hier wird der Vergleich von zwei Codes beschrieben, begrenzt auf nur einige bits.

Code 1: 01 11
Code 2: 10 01

1) Die ersten zwei Bits werden XOR verknüpft:

01 XOR 10 = 11

2) Ergebnis(XOR) wird mit 1 AND verknüpft:

11 AND 01 = 01 → *count* wird um eins erhöht ➔ *count* = 1;

3) Ergebnis(XOR) wird um eins nach rechts geshiftet und mit 1 AND verknüpft:

01 AND 01 = 01 → *count* wird um eins erhöht ➔ *count* = 2;

➔ beide Bits sind unterschiedlich

Dieses wird nun für die nächsten zwei Bits durchgeführt.
Überschreitet *count* die angegebene Fehler Grenze, so wird der Vergleich abgebrochen.

6 Anhang Quellcodes

6.1 Bildschärfe

sharpness.c

```
/*******************************************************************************
*
* DATEINAME
*   sharpness.c
*
*
* BESCHREIBUNG
*   Schärfeindexermittlung mitteles DCT eines
*   Graustufenbildes
*
*
* BENÖTIGTE HEADER (+img62x.lib)
*   sharpness.h
*   fdct_8x8.h
*
*
* FUNKTIONEN
*
*   int
*   sharpness (unsigned char * scaled_buffer)
*
*
* AUTOR
*   Andreas Hilbert /hilbert@cs.tu-berlin.de/
*
*******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>

#include "fdct_8x8.h"        //img62x.lib

#include "sharpness.h"
#include "io.h"              //File-I/O und Bildumwandlung


//*********************************************************//
//*******                                          *******//
//*******   SCHÄRFEFUNKTION MITTELS                *******//
//*******   'fdct_8x8' auf verkleinertem           *******//
//*******   Graustufenbild                         *******//
//*******                                          *******//
//******* scaled buffer:   verkleinertes Bild      *******//
//*******                  =  160*120              *******//
//*******                                          *******//
//*********************************************************//
int sharpness (unsigned char * scaled_buffer) {
        //Rückgabewert
        int             schaerfe=0;

        //Summe über ausgewählte DCT-Koeffizienten und Blöcke
        float           sum=0.0;

        //DCT-Koeffizienten-Blöcke
        short          *dct;

        //Bildorte
        register int    x=0, y=0;

        //8x8-Blocknummern bezogen auf Eingangsbild
        register int    nx=0, ny=0;

        //8x8-Blocknummern bezogen auf DCT-Buffer
        register int    n=0;
        //Block von DCT-Koeffizientenmatrizen allozieren
        dct = (short*)calloc( (8*10*64), sizeof(short) );
```

```
	if(dct==NULL){
		printf("out of memory (dct-blocks)!\n");
		exit(0);
	};


	//Eingangsbildumwandlung (Anforderung von 'fdct_8x8'

	//Blockweises Voranschreiten im Eingangsbild
	//*NY = 160/8 = 20 Blöcke
	//*NX = 120/8 = 15 Blöcke
	//*Bereich in der Bildmitte (mittige 80 von 300 Blöcken)
	for (ny=4;ny<12;ny++) {
		for(nx=5;nx<15;nx++) {

			//Bildrichtiges Umkopieren in 8x8-DCT-Pixel-Block
			for (y=0;y<8;y++) {
				for (x=0;x<8;x++) {

					//zugleich Umwandlung in 'short'
					*(dct + ((ny-4)*10+(nx-5))*64 + y*8 + x )
					=(short)(*(scaled_buffer + (ny*8 + y)*160 + nx*8 + x));
				};
			};
		};
	};

		//*****************************************************//
	//*** DCT direkt auf den arrangierten Bildblöcken ***//
	//*****************************************************//
	fdct_8x8(dct,(8*10));

	//Koeffizientenauswertung
	sum=0.0;

	//Blockweise im DCT-Array
	for (n=0 ; n<(8*10) ; n++) {

		//Aufsummierung der Koeffizienten-Beträge
		//* mittlere Orts-Frequenzen
		//*(3 <= x <= 5) && (3 <= y <=5)
		//*'if'-Schreibweise möglicherweise praktischer

		for (y=3 ; y<6 ; y++) {
			for (x=0 ; x<6 ; x++) {
				sum += fabs((float)(*(dct + n*64 + y*8 + x)));
			};
		};

		for (y=0;y<3;y++) {
			for (x=3;x<6;x++) {
				sum += fabs((float)(*(dct + n*64 + y*8 + x)));
			};
		};
	};

	//Integerumwandlung (Gruppenabsprache)
	schaerfe = (int)(ceil(sum));

	free(dct);

	//Betragsinterpretation erfolgt an anderer Stelle
	return schaerfe;

}
```

6.2 Scaling

scale.c

```
/*****************************************************************************
*
* DATEINAME
*   scale.c
*
* BESCHREIBUNG
*   Skalierung eines Graustufen-Bildes
*   mit Mittelwertfilterung auf 1/16
*   der Originalgröße
*
* HEADER
*   scale.h
*
* FUNKTIONEN
*
*   void
*   scale (unsigned char * buffer, unsigned char * scaled_buffer);
*
*
* AUTOR
*   Andreas Hilbert /hilbert@cs.tu-berlin.de/
*
*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>
#include "scale.h"
#include "io.h"       //File-I/O und Bildumwandlung

//*********************************************************//
//*******                                         *******//
//*******   SKALIERUNGSFUNKTION MIT               *******//
//*******   FESTER GRÖSSENZUORDNUNG UND           *******//
//*******   MITTELWERT-FILTERUNG                  *******//
//*******                                         *******//
//******* buffer:          Originalbild           *******//
//*******                  =  640*480             *******//
//******* scaled buffer:   verkleinertes Bild     *******//
//*******                  =  160*120             *******//
//*******                                         *******//
//*********************************************************//
void scale (unsigned char *buffer, unsigned char* scaled_buffer) {

	//Pixelkoordinaten
	register int		x=0, y=0;
	//Filterbereich
	register int		dx=0, dy=0;
	//Filtersumme
	register float		sum=0.0;

	//Durchlauf im kleinen Bild
	for(y=0;y<120;y++) {
		for(x=0;x<160;x++) {
			//Aufsummierung der zugehörigen Originalpixel (4*4-Block)
			sum=0.0;
			for(dy=0;dy<4;dy++) {
				for(dx=0;dx<4;dx++) {
					//////////////////////////////////////////////////
					sum += (float)(*(buffer + (4*y+dy)*640 + 4*x+dx));
					//////////////////////////////////////////////////
				};
			};
			//Mittelwertzuweisung im kleinen Bild
			sum = sum/16.0;
			*(scaled_buffer + y*160 + x)= (unsigned char)(ceil(sum));
		};
	};
}
```

6.3 Irislokalisierung

locate.h

```
//bildinfo
#define H_SIZE          160
#define V_SIZE          120
#define PIC_SIZE        (H_SIZE*V_SIZE)

//startschrittweite der suchbereichsursprünge (in pixeln)
#define start_step      (int)(ceil( ((double)H_SIZE) / 32.0))

//startauflösung innerhalb des suchbereichs (in pixeln)
#define start_res       (int)(ceil( ((double)H_SIZE) / 32.0))

//minimale x-y-abweichung des gefundenen
//mittelpunkts zwischen 2 funktionsaufrufen
#define precision       (int)(ceil( ((double)H_SIZE) / 160.0))

//maximale anzahl der funktionsaufrufe
#define max_depth       4

//augeninfo
#define iris_radius     (int)(ceil( ((double)H_SIZE) / 3.0))
#define pupil_radius    (int)(ceil( ((double)H_SIZE) / 8.0))
#define EYE_SIZE        (4*pupil_radius*pupil_radius)


//kompakte Übergabe von Koordinaten ermöglichen
struct point {
               int x;
               int y;
               int buf_nr;
};

//kompakte Übergabe von Suchparametern ermöglichen
struct range {
               int x;
               int y;
               int step;
               int res;
};

//********************************//
//********* Prototypen **********//
//********************************//

void
locate_pupil    (unsigned char * buffer, int * mx_pup, int * my_pup);

struct point
max_corr        (unsigned char * buffer, unsigned char * eye,
                 struct point start, struct range search);

void
draw_eye        (unsigned char * eye);
```

locate.c

```
/***************************************************************************
*
* DATEINAME
*   locate.c
*
*
* BESCHREIBUNG
*   Lokalisierung eines Punktes innerhalb der Pupille
*   in einem verkleinertem Graustufen-Augenbild
*   mittels Korrelationsgradmessung
*
*
* BENÖTIGTER HEADER (typendefinitionen & skalierungen)
*   locate.h
```

```
*   -> struct point & struct range
*
*
* FUNKTIONEN
*
*   void
*   locate_pupil (unsigned char * buffer, int * mx_pup, int * my_pup)
*
*   struct point
*   max_corr     (unsigned char * buffer, unsigned char * eye,
*                  struct point start, struct range search)
*
*   void
*   draw_eye     (unsigned char * eye)
*
*
* AUTOR
*   Andreas Hilbert /hilbert@cs.tu-berlin.de/
*
***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>

#include "locate.h"

//*******************************************************//
//*******                                        *******//
//*******    ANFANGSPUNKTBESTIMMUNG FÜR DIE      *******//
//*******    SPÄTERE EXAKTE IRIS-LOKALISATION    *******//
//*******          ( -> iscan.c )                *******//
//*******                                        *******//
//*******    ITERATIVE AUFRUFABFOLGE VON         *******//
//*******    'max_corr'                          *******//
//*******    ZUR UNGEFÄHREN ORTSBESTIMMUNG       *******//
//*******    DES PUPILLENMITTELPUNKTES           *******//
//*******                                        *******//
//*******  buffer: (verkleinertes) Augenbild     *******//
//*******           =  H_SIZE*V_SIZE             *******//
//*******  mx_pup:  X-Koordinate der Pupille     *******//
//*******  my_pup:  Y-Koordinate der Pupille     *******//
//*******                                        *******//
//*******************************************************//
void locate_pupil(unsigned char * buffer, int * mx_pup, int * my_pup) {

       //Pupillenfilter
       unsigned char  *eye;

       //Koordinaten von aktueller Pupille und Vorgänger
       struct point   pupil, pupil_v;

       //Suchbereichsweite und -auflösung
       struct range   search;

       //Suchschritt-Tiefe
       register int   i=0;


       //Erstellen des Pupillenfilters
       eye     =(unsigned char *)calloc(EYE_SIZE, sizeof(unsigned char));
       if(eye==NULL) {
              printf("out of memory (eye-filter)!\n");
              exit(1);
       };

       draw_eye(eye);



       //Initialisierung der Pupillenkoordinaten
       //*Startwert: Bildmitte
       pupil.x        = (int)ceil(  ((double)H_SIZE) / 2.0);
       pupil.y        = (int)ceil(  ((double)V_SIZE) / 2.0);
```

```
        pupil.buf_nr   = pupil.y*H_SIZE + pupil.x;


        //Initialisierung der Suchparameter
        //*Suchbereich: Ganzes Bild - Pupillenradius
        //*Auflösung:   Schrittweite und Filterauflösung
        search.x       = (int)(ceil(  ((double)H_SIZE) / 2.0)) - pupil_radius;
        search.y       = (int)(ceil(  ((double)V_SIZE) / 2.0)) - pupil_radius;
        search.step    =  start_step; //->locate.h
        search.res     =  start_res;  //->locate.h



        //Iterative Suche nach der besten Korerelation
        //zwischen Pupillenfilter und Bild
        do {
                //Merken der zuletzt gefundenen Pupillenkoordinaten
                pupil_v = pupil;


                //*************************************************//
                //*** Suche nach bestkorellierendem Pupillenort ***//
                //*************************************************//
                pupil = max_corr( buffer, eye, pupil, search);

                //Verfeinerung der Suchparameter
                search.x       = (int)(ceil( ((double)search.x    ) / 4.0));
                search.y       = (int)(ceil( ((double)search.y    ) / 4.0));
                search.step    = (int)(ceil( ((double)search.step) / 2.0));
                search.res     = (int)(ceil( ((double)search.res ) / 2.0));

                i++;

        }
        //Abbruch bei erreichter Genauigkeit oder maximaler Schritt-Tiefe
        while(    ( (abs(pupil_v.x - pupil.x) > precision)
                    || (abs(pupil_v.y - pupil.y) > precision) )
                && (i < max_depth)      );

        free(eye);

        //Umrechnen der Koordinaten auf Originalbildgröße (->scale.c - Vereinbarungen)
        *mx_pup                = pupil.x*4;
        *my_pup                = pupil.y*4;

}

//*******************************************************//
//*******                                        *******//
//*******   ERZEUGEN DES PUPILLENFILTERS          *******//
//*******                                        *******//
//*******  eye:  Puillenbild                      *******//
//*******        =  EYE_SIZE                      *******//
//*******        = (2*pupil_radius)^2             *******//
//*******        =  H_SIZE^2/16                   *******//
//*******                                        *******//
//*******************************************************//
void draw_eye(unsigned char * eye) {

        //Koordinaten
        register int   x=0, y=0;

        //Radius
        const int        r=pupil_radius;

        //Fließkommawerte von x, y und Radius
        double           xf=0.0, yf=0.0, rf=0.0;

        //Fließkommawert des Koeffizienten
        double           pixel=0.0;


        rf = (double)r;

        //Quadratisch-radial-abhängige
        //Erzeugung der Werte
```

```
        for ( y = -r ; y < r ; y++ ) {

                for ( x = -r ; x < r ; x++ ) {

                        xf=(double)x;
                        yf=(double)y;

                        pixel = ((xf*xf+yf*yf)/(2*rf*rf));
                        pixel = pixel*pixel;
                        pixel = floor(255.0*pixel);

                        //Zuordnung der Werte auf quadratische Filtermatrix
                        *(eye + ((y+r)*2*r) + (x+r)) = (unsigned char)pixel;
                };
        };

}


//*******************************************************//
//*******                                         *******//
//*******   PUNKT DER MAXIMALEN KORRELATION       *******//
//*******   ZWISCHEN BILD UND PUPILLENFILTER      *******//
//*******                                         *******//
//*******  buffer: (verkleinertes) Eingangsbild   *******//
//*******           =  H_SIZE*V_SIZE              *******//
//*******  eye:     Pupillenbild (Filter)         *******//
//*******  start:   Ausgangspunkt der Suche       *******//
//*******  search:  Filter- und Bereichsauflösung *******//
//*******                                         *******//
//*******************************************************//
struct point max_corr (unsigned char * buffer, unsigned char * eye,
                       struct point start, struct range search) {

        //Rückgabepunkt
        struct point   pupil;

        //Bildkoordinaten
        register int   x=0, y=0;

        //Filterbereichskoordinaten
        register int   dx=0, dy=0;

        //Filterbereichsbegrenzung
        const int      r = pupil_radius;

        //Korrelationssumme und deren Maximum
        register double       k_max = 0.0, k_sum = 0.0;

        //Fließkommawerte der Bild- und Filterpixel
        register double       dp = 1.0, dk = 1.0;


        //Sicherheitsbereich
        pupil.x        = 0;
        pupil.y        = 0;
        pupil.buf_nr   = 0;

        k_max          = 0.0;

        //BILDBEREICH
        //Suchlauf innerhalb eines Bildausschnittes
        //* Ausgangspunkt: 'start.x' und 'start.y'
        //* Auflösung:      'search.step'
        //* Bereich:        'search.y' und 'search.x'
        for ( y = (start.y - search.y) ; y < (start.y + search.y) ; y+=search.step) {

                for ( x = (start.x - search.x) ; x < (start.x + search.x) ; x+=search.step) {

                        //Rücksetzen der Korrelatinssumme
                        k_sum = 0.0;

                        //FILTERBEREICH
                        //Summation über Bereich der Filtermatrix
                        //* Auflösung: 'search.res'
                        for (dy = -r ; dy < r ; dy+=search.res) {
```

```
                        for (dx = -r ; dx < r ; dx+=search.res) {

                                //Überprüfung auf gültige Pixelkoordinaten
                                //* Bereichsüberschreitung bei korrektem
                                //* Funktionsaufruf nicht möglich
                                if( ((y+dy)>0) && ((y+dy)<V_SIZE) && ((x+dx)>0)
                                    && ((x+dx)<H_SIZE) ) {

                                        //Fließkommaumwandlung und
                                        //Mittelwertbefreiung der
                                        //Bild- und Filterpixel
                                        dp = (double)(*(buffer + (y+dy)*H_SIZE + (x+dx)))
                                                - 127.0;
                                        dk = (double)(*(eye +    (r+dy)*2*r    + (r+dx)))
                                                - 127.0;
                                }
                                else {
                                        dp = 0.0;
                                        dk = 0.0;
                                };

                                //************************//
                                //*** Korrelationssumme ***//
                                //************************//
                                k_sum += dp*dk;

                        };
                };
                //ENDE FILTERBEREICH

                //Überprüfung und Aktualisierung
                //der größter Summe
                if (k_sum > k_max) {
                        k_max           =  k_sum;
                        pupil.x         =  x;
                        pupil.y         =  y;
                        pupil.buf_nr    = (y*H_SIZE)+x;
                };
        };
    };
    //ENDE BILDBEREICH
    //Rückgabe des Punktes mit der größten
    //Korrelationssumme
    return pupil;
}
```

iscan.h

```
int  m_v        (int i, int j);


void kreis      (int ax, int ay, int bx, int by, int cx, int cy, int * mitte_x, int * mitte_y,
int * radius);


void locate     (unsigned char * bild, int ax, int ay, int * mitte_x, int * mitte_y, int
*radius, int iris, int rad_pup);


int  cmp        (const void *e_1, const void *e_2);


void iscan_main (unsigned char *bild,  int *mx_pup, int *my_pup, int *r_pup, int *r_iris);
```

iscan.c

```
/******************************************************************************
*
* DATEINAME
*   iscan.c
*
*
```

```
* BESCHREIBUNG
*   Genaue Iris-Lokalisation (Mittelpunkte, Radien) aus
*       ungefährem Anfangspunkt in Graustufen-Augenbild;
*       einige zugehörige Hilfsroutinen
*
*
* REQUIRED HEADER (+img62x.lib)
*       threshold.h
*       iscan.h
*
*
* FUNKTIONEN
*
*       void
*       kreis          (int ax, int ay, int bx, int by, int cx, int cy, int * mitte_x, int *
mitte_y, int * radius) {
*
*       void
*       punkt_zeichnen (unsigned char * bild, int ax, int ay){
*
*       void
*       kreis_zeichnen (unsigned char * bild, int mx, int my, int r){
*
*       int
*       m_v            (int x, int y)
*
*       void
*       locate         (unsigned char * bild, int ax, int ay, int * mitte_x, int * mitte_y, int
*radius, int iris, int rad_pup){
*
*       int
*       cmp            (const void *e_1, const void *e_2)
*
*       void
*       iscan_main     (unsigned char *bild,  int *mx_pup, int *my_pup, int *r_pup, int
*r_iris)
*
*
*
* AUTOREN
*   Juan Jose Burred  /jjburred@inicia.es/
*       Marcel Roth                    /romabghj@linux.zrz.tu-berlin.de/
*
*******************************************************************************/

#define _TI_ENHANCED_MATH_H 1  //um "round" zu benutzen
#define pi 3.141592654

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>

#include "iscan.h"
#include "threshold.h"     //img62x.lib

//************************************************************//
//*******                                             *******//
//*******   KREISMITTELPUNKT- UND RADIUS-             *******//
//*******   BESTIMMUNG AUS 3 PUNKTEN MITTELS          *******//
//*******   DER CRAMERSCHEN REGEL                     *******//
//*******                                             *******//
//******* ax, ay:     x-y-Koordinaten Punkt A         *******//
//******* bx, by:     x-y-Koordinaten Punkt B         *******//
//******* cx, cy:     x-y-Koordinaten Punkt C         *******//
//*******                                             *******//
//******* mitte_x:  x-Koordinate des Mittelpunkts     *******//
//******* mitte_y:  y-Koordinate des Mittelpunkts     *******//
//******* radius:   Radius des Kreises                *******//
//*******                                             *******//
//************************************************************//
void kreis(int ax, int ay, int bx, int by, int cx, int cy, int * mitte_x, int * mitte_y, int *
radius) {
```

```
        //Seitenmittelpunkte
        double m1x, m1y, m2x, m2y;

        //Richtungsvektoren
        double rs1x, rs1y, rs2x, rs2y;

        //Gleichungssystem
        double det,t;

        //Mittelpunkte der Seiten des Dreiecks
        m1x = ((double)ax+(double)bx)/2;
        m1y = ((double)ay+(double)by)/2;
        m2x = ((double)bx+(double)cx)/2;
        m2y = ((double)by+(double)cy)/2;


        //Richtungsvektoren der Seiten
        rs1x = bx-ax;
        rs1y = by-ay;
        rs2x = cx-bx;
        rs2y = cy-by;


        //Cramersche Regel (Lösung des LGS)
        det = (-rs1y)*(-rs2x)+(-rs1x)*rs2y;
        t   = ((-rs2x)*(m1x-m2x)+rs2y*(m2y-m1y))/det;


        //Bestimmung der Mitte und des Radius
        *mitte_x = round(m1x+t*rs1y);
        *mitte_y = round(m1y+t*(-rs1x));
        *radius  = round(sqrt(pow((*mitte_x-ax),2)+pow((*mitte_y-ay),2)));
}

//*********************************************************//
//*******                                          *******//
//*******   WEISSES MARKIEREN EINES PUNKTES        *******//
//*******                                          *******//
//******* bild:     Eingangs-Graustufenbild        *******//
//******* ax, ay:   x-y-Koordinaten des Punktes    *******//
//*******                                          *******//
//*******           640*480                        *******//
//*******                                          *******//
//*********************************************************//
void punkt_zeichnen(unsigned char * bild, int ax, int ay){

        *(bild + m_v(ax,ay) ) = 255;  //  255=weiss

}


//*************************************************************//
//*******                                              *******//
//*******   ZEICHNEN EINES WEISSEN KREISES             *******//
//*******                                              *******//
//******* bild:     Eingangs-Graustufenbild            *******//
//******* mx, my:   x-y-Koordinaten des Mittelpunktes  *******//
//******* r:       Radius des Kreise                  *******//
//*******                                              *******//
//*************************************************************//
void kreis_zeichnen(unsigned char * bild, int mx, int my, int r){


        //Kreispunkte, Winkel
        register int i, px, py;
        register double phi=0;

        //Schrittweiten
        double          schritte   = 2*pi*r;
        register double schritt_gr = 2*pi/schritte;

                //Einzeichnen der weissen Pixel
        for (i=1 ; i<=schritte ; i++){

                px = mx + floor(r*cos(phi));
                py = my + floor(r*sin(phi));
```

```
                *(bild+m_v(px,py))=255;

                phi+=schritt_gr;
        }

}

//************************************************************//
//*******                                             *******//
//*******   INDEX-UMWANDLUNG VON X-Y-KOORDINATEN      *******//
//*******   (BILD AUF DEM MONITOR)                    *******//
//*******   ZU ELEMENTNUMMER IN EINEM VEKTOR          *******//
//*******   (BILD IM SPEICHER)                        *******//
//*******                                             *******//
//*******  x, y:  x-y-Bildkoordinaten                 *******//
//*******                                             *******//
//*******         640*480                             *******//
//*******                                             *******//
//************************************************************//
int m_v(int x, int y ) {


        //Index-Nummer = Zeilenbreite * Zeile  +  Spalte
        return (640*y + x);

}

//************************************************************//
//*******                                             *******//
//*******   BETRAG EINES ORTSVEKTORS                  *******//
//*******                                             *******//
//*******  x, y:  x-y-Ortskoordinaten                 *******//
//*******                                             *******//
//************************************************************//
double betrag(int x, int y) {


        return (sqrt( x*x + y*y ) );

}

//*************************************************************//
//*******                                              *******//
//*******   KREIS-LOKALISIERUNG DURCH SUCHE            *******//
//*******   NACH PUNKTETRIPELN AN SCHWARZ-WEISS-       *******//
//*******   KANTEN IN THRESHOLD-BILDERN                *******//
//*******                                              *******//
//******* ax, ay:   x-y-Startkoordinaten               *******//
//******* mitte_x:  x-Resultat des Mittelpunkts        *******//
//******* mitte_y:  y-Resultat des Mittelpunkts        *******//
//******* radius:   Resultat für den Kreisradius       *******//
//******* iris:     Flag für Iris-Aussenkreis-Suche *******//
//******* rad_pup:  Startradius für Pupille            *******//
//*******                                              *******//
//*******           640*480                            *******//
//*******                                              *******//
//*************************************************************//
void locate(unsigned char * bild, int ax, int ay, int * mitte_x, int * mitte_y, int * radius,
int iris, int rad_pup){


        //Länge des "Sicherheitsbereiches"
        int max = 25;

        //Zähler
        register int i, j;

        //Anfangsrichtungsvektor
        double rv1, rv2;

        //Punktkoordinaten und Verschiebung
        int px, py, abstand1, abstand2;

        //Beträge und Zwischenwerte
        double bet, bet2, temp, temp1, temp2;
```

```
//Pixelzähler und Bereichsflags
register int summe, bsumme, flag, on, weiter;

//Kreis-Zwischenergebnisse
int mx, my, r;

//3D-Matrix zur Speicherung der Grenzenpunkte
int ziel[6][3][2];

// Matrizen für die Median-Berechnung
int matrix_r[6], matrix_x[6], matrix_y[6];


*mitte_x = 0;
*mitte_y = 0;
*radius  = 0;

for (j=0 ; j<=5 ; j++) {


        //6 verschiedene Anfangsrichtungsvektoren (6 Drehungen der 3 Achsen)

        if        (j==0) {
                rv1 =  1;
                rv2 =  0;

        } else if (j==1) {
                rv1 =  1;
                rv2 =  1;

        } else if (j==2) {
                rv1 =  1;
                rv2 =  2;

        } else if (j==3) {
                rv1 = -1;
                rv2 =  3;

        } else if (j==4) {
                rv1 =  1;
                rv2 =  4;

        } else if (j==5) {
                rv1 = -5;
                rv2 =  1;
        }


        //Um den Algorithmus zu optimieren, müssen wir einen Richtungsvektor
        //der Länge 1 (Einheitsvektor) benutzen.
        bet = betrag(rv1, rv2);
        rv1 = rv1/bet;
        rv2 = rv2/bet;


        //3 zueinander senkrechten Achsen
        for (i=0 ; i<=2 ; i++) {

                px = ax;
                py = ay;

                temp1 = px;
                temp2 = py;

                summe = bsumme = 0;

                flag=1;
                weiter=1;
                on=0;

                //Pixelweise in Vektorrichtung
                while (weiter) {

                        if (iris==0 || on==1) {
```

```
                        if ( *(bild + m_v(px,py)) != 0 ) {

                                //Summe der weissen Pixel
                                summe += 1;

                                if ( flag  &&  (bsumme > max) ) {

                                        //Rücksetzten der Flag nach Erkennung
                                        flag = 0;

                                        //Speicherung der Kantenkoordinaten
                                        ziel[j][i][0] = px;
                                        ziel[j][i][1] = py;
                                }

                        }
                else {

                        //Summe der schwarzen Pixel
                        bsumme++;

                        if (summe > 0) {

                                flag  = 1;
                                summe = summe-1;
                        }

                }


                //Abbruchkriterien für die Suche
                if (((summe > max)  &&  (bsumme > max))  ||  (px < 1)  ||  (px >
640)  ||  (py < 1) || (py > 480))
                        weiter = 0;
                }


                //Berechnung der nächsten Position auf der Erkennungsrichtung

                temp1 = temp1 + rv1;
                temp2 = temp2 + rv2;

                px = round(temp1);
                py = round(temp2);



                //Berechnung des Abstands zwischen Erkennungspunkt und Mittelpkt

                abstand1 = px - ax;
                abstand2 = py - ay;
                bet2     = betrag(abstand1, abstand2);


                //Wenn wir die Iris Erkennen, lesen wir die Pixels nur ab dem
                //doppelten Radius der Pupille

                if ( (iris==1)  &&  (bet2 > 2*rad_pup) ) {
                        on=1;
                }


        }
        // Ende der while-Schleife


        //Berechnung des nächsten Vektors, senkrecht zum vorigen

        temp =  rv1;
        rv1  =  rv2;
        rv2  = -temp;
}

//Aufruf der Kreisberechnungsfunktion
```

```
        kreis(ziel[j][0][0], ziel[j][0][1], ziel[j][1][0], ziel[j][1][1], ziel[j][2][0],
ziel[j][2][1], &mx, &my, &r);

        //Kreisparameter(r, x, y) in Matrix schreiben

        matrix_r[j]=r;
        matrix_x[j]=mx;
        matrix_y[j]=my;

        }
        //Ende der Schleife für 6 Kreise



        //** "Median"-Berechnung

        //Werte mit Standard-Qicksort nach Grösse sortieren

        qsort(  (void*)(&matrix_r[0]) , 6 , sizeof(int), cmp);
        qsort(  (void*)(&matrix_x[0]) , 6 , sizeof(int), cmp);
        qsort(  (void*)(&matrix_y[0]) , 6 , sizeof(int), cmp);

        //Mediane der einzelnen Parameter (gerade Anzahl)

        *mitte_x = (int) ((float) matrix_x[2] + (float) matrix_x[3])/2.0;
        *mitte_y = (int) ((float) matrix_y[2] + (float) matrix_y[3])/2.0;
        *radius  = (int) ((float) matrix_r[2] + (float) matrix_r[3])/2.0;

}

//************************************************************//
//*******                                           *******//
//*******   VERGLEICHSFUNKTION FÜR QUICKSORT-       *******//
//*******   AUFRUFE                                 *******//
//*******                                           *******//
//******* e1, e2:   Listenelementadressen           *******//
//*******                                           *******//
//************************************************************//
int cmp(const void * e_1, const void * e_2) {

        //vergleicht die übergebenen Listenelemnte
        //als Integer-Werte
        return ( *((int*)e_1) - *((int*)e_2));
}


//************************************************************//
//*******                                           *******//
//*******   EXAKTE IRIS-LOKALISIERUNG DURCH         *******//
//*******   BESTIMMUNG DES MITTELPUNKTES UND DER    *******//
//*******   INNEN- UND AUSSENRADIEN MITTELS         *******//
//*******   2 VERSCHIEDENER THRESHOLD-BILDER        *******//
//*******                                           *******//
//******* bild:     das Augenbild                   *******//
//******* mx_pup:   x-Resultat des Mittelpunkts     *******//
//******* my_pup:   y-Resultat des Mittelpunkts     *******//
//******* r_pup:    Resultat für den Innenradius    *******//
//******* r_iris:   Resultat für den Aussenradius   *******//
//*******                                           *******//
//*******           640*480                         *******//
//*******                                           *******//
//************************************************************//
void iscan_main(unsigned char * bild,  int * mx_pup, int * my_pup, int * r_pup, int * r_iris)
{


        //Threshold Bild
        unsigned char * buffer_2;

        // Threshold Werte
        int pupille_thresh =  90;
        int iris_thresh    = 160;

        //Debug-Koordinaten für zweiten Mittelpunkt
        int mx_iris, my_iris;
```

```
        // Anfangspunkt
        int ax = *mx_pup;
        int ay = *my_pup;


        // Buffer für die Thresholdbilder allozieren
        buffer_2 = (unsigned char *)calloc(640*480, sizeof(unsigned char) );
        if(buffer_2==NULL) {
                printf("out of memory (threshold-pic)!\n");
                exit(1);
        };



        //******************** PUPILLE *************************************
        //*                                                                 *
        //*     Kontrastbild, dann Mittelpkt & Radius der Pupille bestimmen *
        //*                                                                 *
        //******************************************************************

    //der Pupillen-Bereich bleibt als schwarze Fläche übrig (img62x.lib)
        threshold((unsigned char *)bild, (unsigned char *)buffer_2, 640, 480, (unsigned
char)pupille_thresh);

        locate(buffer_2, ax, ay, mx_pup, my_pup, r_pup, 0, 0);


        //******************* IRIS *****************************************
        //                                                                 *
        // Kontrastbild, dann Mittelpkt & Radius der Iris bestimmen       *
        //                                                                 *
        //******************************************************************

    //der Iris-Bereich bleibt als schwarze Fläche übrig (img62x.lib)
        threshold((unsigned char *)bild, (unsigned char *)buffer_2, 640, 480, (unsigned
char)iris_thresh);

        locate(buffer_2, ax, ay, &mx_iris, &my_iris, r_iris, 1, *r_pup);


        free(buffer_2);
}
```

6.4 Programmstruktur

6.4.1 Hauptprogramm - *MAIN.c*

```
/*****************************************************************************
*
* DATEINAME
*       MAIN.c
*
* BESCHREIBUNG
*       unfertige Hauptfunktion
*       RTDX nicht implementiert (funktioniert nicht mit Videoboard)
*
* HEADER
*       video_main.h / io.h
*       scale.h
*       sharpness.h
*       filter.h
*       GetScanCode.h
*
* LIBRARIES
*       img62x.lib
*       rtdx.lib
*       rts6201.lib
*
* REMARKS
*       heap-size 0x100000
*
* AUTOR
*       Signalprozessor-Labor TU-Berlin
```

```
*       Gruppe: frvo 2001/2002
*       http://ntife.ee.tu-berlin.de/labore/dsp/frvo/index.html
*
* ZUSAMMENSTELLUNG
*       Andreas Hilbert /hilbert@cs.tu-berlin.de/
*
********************************************************************************/

//Schwellwert für scharfes/unscharfes Bild
#define sharpness_threshold 5500

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>
#include "c6211dsk.h"

#include "video_main.h"    //Videochip-Steuereung
#include "io.h"            //File-I/O, Bildumwandlung
#include "GetScanCode.h"   //Bildanalyse
#include "scale.h"         //Bildskalierung
#include "sharpness.h"     //Schärfemessung
#include "filter.h"        //Gabor-Filterung (hier für die Tabellen)

//**************************************************//
//************                 ********************//
//************  HAUPTFUNKTION  ********************//
//************                 ********************//
//**************************************************//
int main(int argc, char * * argv) {

        //512 BYTE - IRIS-CODE
        unsigned char  * arrayPointer;

        //AUGENBILD
        unsigned char  * buffer;

        //VERKLEINERTES AUGENBILD (SHÄRFEMESSUNG UND IRIS-LOKALISATION)
        unsigned char  * scaled_buffer;

        //GABOR-TABELLEN FÜR FILTERUNG
        double         * regabtab;
        double         * imgabtab;

        //SCHÄRFE-INDEX
        int              sharp;

        printf("starting programm...\n\n");

        //*******************************************************//
        //*******                                         *******//
        //*******   INITIALISIERUNG DES VIDEOBOARDS       *******//
        //*******                                         *******//
        //*******************************************************//

        StartVideo();


        //**********************************************************//
        //*******                                            *******//
        //*******  EINMALIGE, GLOBALE SPEICHERALLOKATION     *******//
        //*******  FÜR DAS ORIGINAL DES AUGEN-BILDES UND     *******//
        //*******  DES IRIS-CODES                            *******//
        //*******                                            *******//
        //**********************************************************//

        arrayPointer   = (unsigned char*)calloc((2*32*8),  sizeof(unsigned char));
        buffer         = (unsigned char*)calloc((640*480), sizeof(unsigned char));
        if ((buffer == NULL) || (arrayPointer == NULL) ) {
               printf("out of memory (pic or code) !\n");
               exit(1);
        };

        //**********************************************************//
        //*******                                            *******//
        //*******   EINMALIGES, GLOBALES ERZEUGEN            *******//
```

```
//*******   DER GABORTABELLEN FÜR SPÄTERES,           *******//
//*******   WIEDERHOLTES FILTERN                      *******//
//*******                                             *******//
//*************************************************************//

printf("#init gabortabs...\n");
regabtab = (double *)calloc(GABTABSIZE*TABELLEN, sizeof(double));
imgabtab = (double *)calloc(GABTABSIZE*TABELLEN, sizeof(double));
if ((regabtab == NULL) || (imgabtab == NULL)) {
       printf("out of memory (gabortabs) !\n");
       exit(1);
};
init_gabtab(regabtab, imgabtab);
printf("result(init gabortabs)...done.\n\n");

//**********************************************************//
//*******                                       *******//
//*******  AB HIER MÜSSTE DIE WIEDERHOLUNGS-     *******//
//*******  SCHLEIFE DES RTDX-HAUPTPROGRAMMS      *******//
//*******  BEGINNEN                              *******//
//*******                                       *******//
//**********************************************************//

//*********  ANFANG DER RTDX-SCHLEIFE  *********//

       //**********************************************************//
       //*******                                       *******//
       //*******  ZUNÄCHST WIRD SOLANGE EIN BILD        *******//
       //*******  AUFGENOMMEN, BIS DIESES SHARF IST     *******//
       //*******                                       *******//
       //**********************************************************//

       scaled_buffer  = (unsigned char *)calloc((160*120), sizeof(unsigned char));
       if ( scaled_buffer == NULL ) {
              printf("out of memory (scaled pic) !\n");
              exit(1);
       };

       do {
              //**********************************************************//
              //*******                                       *******//
              //*******  EIN BILD WIRD AUS DEN FIFOS DES       *******//
              //*******  VIDEOBOARDS AUSGELESEN                *******//
              //*******                                       *******//
              //**********************************************************//

              printf("#get image...\n");
              GetImage(buffer);
              printf("result(get image)...done.\n\n");

              //**********************************************************//
              //*******                                       *******//
              //*******  ES WIRD EINE STARK VERKLEINERTE       *******//
              //*******  VERSION DES BILDES ERZEUGT            *******//
              //*******                                       *******//
              //**********************************************************//

              printf("#scale image...\n");
              scale(buffer, scaled_buffer);
              printf("result(scale image)...done.\n\n");

              //**********************************************************//
              //*******                                       *******//
              //*******  MITTELS DCT AUF DEM SKALIERTEN BILD   *******//
              //*******  WIRD DIE SCHÄRFE ERMITTELT            *******//
              //*******                                       *******//
              //**********************************************************//

              printf("#measure sharpness...\n");
              sharp = sharpness(scaled_buffer);
              if(sharp < sharpness_threshold)
                     printf("result(measure sharpness): sharpness = %d
                            : NOT SHARP\n\n", sharp);
              else   printf("result(measure sharpness): sharpness = %d
                            : SHARP    \n\n", sharp);
       }
```

```
            while (sharp < sharpness_threshold);

            //*******************************************************//
            //*******                                        *******//
            //*******  IST EIN SCHARFES BILD AUFGENOMMEN      *******//
            //*******  WORDEN, SO BEGINNT NUN DIE ERMITTLUNG  *******//
            //*******  DES IRIS-CODES                         *******//
            //*******                                        *******//
            //*******************************************************//

            printf("#get scan code...\n\n");
            GetScanCode(regabtab, imgabtab, buffer, scaled_buffer, arrayPointer);
            printf("result(get scan code)...done.\n\n");

            //*******************************************************//
            //*******                                        *******//
            //*******  NUN KÖNNTE DIE AUSWERTUNG DER          *******//
            //*******  GEWONNENEN DATEN ERFOLGEN              *******//
            //*******                                        *******//
            //*******************************************************//

        //*********  ENDE DER RTDX-SCHLEIFE  ***********//

        //*******************************************************//
        //*******                                        *******//
        //*******  POWER DOWN MODE DES VIDEOBOARDS        *******//
        //*******  UND FREIGABE DER SPEICHERBEREICHE      *******//
        //*******                                        *******//
        //*******************************************************//

        StopVideo();

        free(regabtab);
        free(imgabtab);
        free(buffer);
        free(arrayPointer);

        printf("\n\nProgramm successfuly completed.\n");
        return 0;
}
```

6.4.2 Code-Gewinnung - *GetScanCode.c*

```
/*******************************************************************************
*
* DATEINAME
*   GetScanCode.c
*
* BESCHREIBUNG
*   Ermittlung des Iris-Codes, Zusammenfassung
*   der Bildanalyse-Algorithmen
*
* HEADER
*   GetScanCode.h
*   locate.h
*   iscan.h
*   poltrans.h
*   filter.h
*   icode.h
*
* FUNKTION
*
*   void
*   GetScanCode (double *regabtab, double *imgabtab, unsigned char *buffer,
*                unsigned char *scaled_buffer, unsigned char *code)
*
* AUTOR
*   Andreas Hilbert /hilbert@cs.tu-berlin.de/
*
*******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "c6x.h"
```

```
#include "c6211dsk.h"

#include "io.h"              //File-I/O
#include "locate.h"          //Anfangspunktbestimmung
#include "iscan.h"           //Dimensionsbestimmung
#include "poltrans.h"        //Polarkoordinatentransformation
#include "filter.h"          //Gabor-transformation
#include "icode.h"           //Code-Quantisierung


//***************************************************************//
//*******                                               *******//
//*******   ZUSAMMENFASSUNG DER BILDANALYSE/            *******//
//*******   DER ENTSPRECHENDEN FUNKTINSAUFRUFE/         *******//
//*******   ERMITTLUNG EINES IRIS-CODES                 *******//
//*******                                               *******//
//******* regabtab:       Gabor-Koeffizeienten-Tabelle  *******//
//*******                 =  GABTABSIZE*TABELLEN        *******//
//******* imgabtab:       Gabor-Koeffizeienten-Tabelle  *******//
//*******                 =  GABTABSIZE*TABELLEN        *******//
//******* buffer:         das Augenbild                 *******//
//*******                 =  640*480                    *******//
//******* scaled_buffer:  das verkleinerte Augenbild    *******//
//*******                 =  160*120                    *******//
//******* code:           der Iris-Code                 *******//
//*******                 =  512                        *******//
//*******                                               *******//
//***************************************************************//
void GetScanCode(double *regabtab, double *imgabtab, unsigned char *buffer,
                       unsigned char *scaled_buffer, unsigned char *code) {

       //polartransformierte Iris
       unsigned char  *pol_buffer;

       //gaborgefilterte Iris
       double *re_buffer;
       double *im_buffer;

       //Iris-Abmessungen (Radien und Mittelpunkte)
       int     mx_pup=0, my_pup=0, r_pup=0, r_iris=0;
       //***********************************************************//
       //*******                                           *******//
       //*******  DURCH KORRELATIONSMESSUNG WIRD AUF       *******//
       //*******  DEM VERKLEINERTEN BILD DIE UNGEFÄHRE     *******//
       //*******  LAGE DER PUPILLE UND DAMIT DES           *******//
       //*******  IRIS-MITTELPUNKTS BESTIMMT               *******//
       //*******                                           *******//
       //***********************************************************//

       printf("#locate pupil...\n");
       locate_pupil(scaled_buffer, &mx_pup, &my_pup);
       printf("result(locate pupil): mx=%d my=%d\n\n", mx_pup, my_pup);


       //***********************************************************//
       //*******                                           *******//
       //*******  MITTELS ZWEIER THRESHOLD_BILDER UND      *******//
       //*******  DER CRAMERSCHEN REGEL WIRD NUN DER       *******//
       //*******  GENAUE IRIS-MITTELPUNKT, IHR INNEN-      *******//
       //*******  UND AUSSENRADIUS BESTIMMT                *******//
       //*******                                           *******//
       //***********************************************************//

       printf("#get iris dimension...\n");
       iscan_main (buffer, &mx_pup, &my_pup, &r_pup, &r_iris);
       printf("result(get iris dimension): mx=%d my=%d r_pup=%d r_iris=%d\n\n",
               mx_pup, my_pup, r_pup, r_iris);


       //***********************************************************//
       //*******                                           *******//
       //*******  UM SPÄTERE RECHNUNGEN ZU VEREINFACHEN    *******//
       //*******  BZW. ZU ERMÖGLICHEN, WIRD EINE POLAR-    *******//
       //*******  KOORDINATENTRANSFORMATION DES IRIS-      *******//
       //*******  RINGS DURCHGEFÜHRT; MAN ERHÄLT EIN       *******//
       //*******  RECHTECKIGES ABBILD EINES GROSSTEILS     *******//
```

```
        //*******  DER IRIS (KRITISCHE REFLEXIONSBEREICHE *******//
        //*******  WERDEN DABEI AUSGELASSEN)              *******//
        //*******                                        *******//
        //*******************************************************//

        printf("#polartransform...\n");
        pol_buffer = (unsigned char*)calloc(512*128, sizeof(unsigned char) );
        if(pol_buffer==NULL){
                printf("out of memory (pol_buffer)!\n");
                exit(0);
        };
        poltrans (mx_pup, my_pup, r_pup, r_iris, buffer, pol_buffer);
        printf("result(polartransform)...done.\n\n");


        //*******************************************************//
        //*******                                        *******//
        //*******  MITTELS DER ERSTELLTEN GABORTABELLEN   *******//
        //*******  WIRD DAS IRIS-ABBILD NUN EINER 2D-     *******//
        //*******  KOMPLEXEN GABORFILTERUNG UNTERZOGEN;   *******//
        //*******  WIR ERHALTEN 2 BILDER, DIE REAL- UND   *******//
        //*******  IMAGINÄRTEIL DES SO GEFILTERTETN       *******//
        //*******  IRIS-RINGS REPRÄSENTIEREN              *******//
        //*******                                        *******//
        //*******************************************************//

        printf("#gaborfilter...\n");
        re_buffer = (double*)calloc(64*16, sizeof(double) );
        im_buffer = (double*)calloc(64*16, sizeof(double) );
        if ((re_buffer == NULL) || (im_buffer == NULL)) {
                printf("out of memory (gaborpics) !\n");
                exit(1);
        };
        filter (512, 128, regabtab, imgabtab, pol_buffer, re_buffer, im_buffer);
        free(pol_buffer);
        printf("result(gaborfilter)...done.\n\n");
        //*******************************************************//
        //*******                                        *******//
        //*******  AUS DEN ERGEBNISSEN DER GABORFILTERUNG *******//
        //*******  WIRD NUN DER IRISCODE GEWONNEN, WOBEI  *******//
        //*******  DIE BITS JEWEILS NACH 2-BIT QUANTI-    *******//
        //*******  SIERTER PHASENLAGE DES FILTERAUSGANGS  *******//
        //*******  GESETZT WERDEN                         *******//
        //*******                                        *******//
        //*******************************************************//

        printf("#get icode...\n");
        icode (re_buffer, im_buffer, code);

        free(re_buffer);
        free(im_buffer);
        printf("result(get icode)...done.\n\n");

}
```

6.4.3 Linker Command File - *GetScanCode_lnk.cmd*

```
/*
 *  Copyright 1998 by Texas Instruments Incorporated.
 *  All rights reserved. Property of Texas Instruments Incorporated.
 *  Restricted rights to use, duplicate or disclose this code are
 *  granted through contract.
 */

-c
-heap  0x100000

-stack 0x100000
-u __vectors
-u _auto_init

_HWI_Cache_Control = 0;
```

```
_RTDX_interrupt_mask = ~0x000001808;

MEMORY
{
       VECS:   o = 00000000h         l = 00000200h  /* interrupt vectors*/
       PMEM:   o = 00000200h         l = 0000FE00h  /* Internal RAM (L2) mem*/

       SDRAM:  origin = 0x80000000,  len = 0x400000
}

SECTIONS
{
       .intvecs        > 0h
       .vectors        > SDRAM
       .text           > SDRAM
       .rtdx_text      > SDRAM
       .bss            > SDRAM
       .cinit          > SDRAM
       .const          > SDRAM
       .far            > SDRAM
       .stack          > SDRAM
       .cio            > SDRAM
       .sysmem         > SDRAM
       .data           > SDRAM
       .rtdx_data      > SDRAM
       .switch         > SDRAM
       .pinit          > PMEM
}
```

6.5 Targetapplikation

Im folgenden Quellcode sind die Routinen für die Berechnung des Iris Codes etc. nicht vorhanden.

```
/***********************+******************************************
 *                                                                      *
 *            Target Applikation                                        *
 *            DSP-Labor 2001/02                                         *
 *                                                                      *
 *            Fazli Ayan, Hakan Uluc, Hikmet Citak               *
 *                                                                      *
 *            Letzte Änderung: 10.02.2002                        *
 *                                                                      *
 ******************************************************************/


// Einfügen der Include - Dateien
#include <rtdx.h>        // defines RTDX target API calls
#include "..\target.h"   // defines TARGET_INITIALIZE()
#include <stdio.h>       // C_I/O


// Makros
#define DATAVOLUME 512


// ****************** Kanal Deklarationen ****************** //
RTDX_CreateOutputChannel( ochan );
RTDX_CreateInputChannel( ichan );
RTDX_CreateOutputChannel( ochan_SAFEARRAY );


// ***************** Pointer Deklarationen ***************** //
RTDX_output_channel *pochan;
RTDX_input_channel *pichan;
RTDX_output_channel *pochan_SAFEARRAY;


// ******* RTDX Initialisierung und Kanalaktivierung ******* //
void init_RTDX()
{
```

```
    TARGET_INITIALIZE();
    RTDX_enableInput( pichan );
    RTDX_enableOutput( pochan );
    RTDX_enableOutput( pochan_SAFEARRAY );
    printf("Kanäle wurden aktiviert ... \n");
}


// ***** RTDX Deinitialisierung und Kanaldeaktivierung ****** //
void deinit_RTDX()
{
       RTDX_disableInput( pichan );
       RTDX_disableOutput( pochan );
       RTDX_disableOutput( pochan_SAFEARRAY );
       printf("Kanäle wurden deaktiviert ... \n");
}


//  Ganze Zahl von der Hostapplikation über pichan empfangen  //
short int_empfangen()
{
       int status;
       short toReceive;

       status = RTDX_read( pichan , &toReceive , sizeof(toReceive) );
       if ( status == 0 )
       {
              puts( "ERROR: RTDX_read failed!\n" );
              exit( -1 );
       }
       if (toReceive == 5)
              printf("REQ empfangen ...\n");
       else if (toReceive == 10)
              printf("Iris wird aufgenommen ...\n");

       return toReceive;
}


// ** Ganze Zahl an die Hostapplikation über pochan senden ** //
void int_senden(short toSend)
{
       int status;

       status = RTDX_write( pochan, &toSend, sizeof(toSend) );
       if (toSend == 10)
              printf("ACK gesendet ...\n");

       if ( status == 0 )
       {
              puts( "ERROR: RTDX_write failed!\n" );
              exit( -1 );
       }
       while ( RTDX_writing != NULL )
       {
#if RTDX_POLLING_IMPLEMENTATION
                     RTDX_Poll();
#endif
       }
}


// *** iCode über pochan_SAFEARRAY an die Hostapp. senden *** //
void array_senden(unsigned char *array)
{
       int status;

       status = RTDX_write( pochan_SAFEARRAY , array , DATAVOLUME);

       if ( status == 0 )
       {
           puts( "ERROR: RTDX_write failed!\n" );
           exit( -1 );
       }

       while ( RTDX_writing != NULL )
```

```
        {
#if RTDX_POLLING_IMPLEMENTATION
                        RTDX_Poll();
#endif
        }
        printf("Iris Code gesendet ... \n" );

}



void main()
{

// *********************** Iris Code ********************* //
                unsigned char arrayPointer[DATAVOLUME];


        short i;
        short Command;

// Adresszuweisung
         pochan                         = &ochan;
         pichan                                 = &ichan;
                pochan_SAFEARRAY        = &ochan_SAFEARRAY;

// Initialisierung und Aktivierung
         init_RTDX();




// Füllen des Arrays mit einem Bsp. Code
        for ( i=0 ;i<DATAVOLUME ; i++)
        {
                arrayPointer[i] = i+10;
        }

        printf("Beispiel Iris Code geschrieben ...\n");


        printf("Bereit, um das Iris zu erfassen ...\n");


// Request empfangen
                Command = int_empfangen();


if (Command != 20)
{
        // Acknowledge senden
                Command = 10;
                        int_senden( Command );


        // Request empfangen, um weiterzumachen
                        Command = int_empfangen();


        // Code an die Hostapplikation senden
                        array_senden(arrayPointer);

} // Ende if-Anweisung



// Deinitialisierung
                deinit_RTDX();

                printf("Programm erfolgreich abgeschlossen!\n");
}
```

6.6 Hostapplikation

Die Hostapplikationen besteht aus drei Quelldateien, eine für das Auswahlmenü, eine für die Funktionen und eine Datei für das RTDX.

HostMain.cpp: Diese Datei beinhaltet das Auswahlmenü

```
/************************************************************
*                                                                   *
*       Mainprogram for iScan                            *
*       DSP-Labor 2001/2002                              *
*                                                                   *
*       Fazli Ayan                                                  *
*       Hakan Uluc                                                  *
*       Hikmet Citak                                                *
*                                                                   *
*       Last change: 10.02.2002                                     *
*                                                                   *
*************************************************************/


#import        "..\rtdxint.dll"
#include <iostream.h>
#include <conio.h>
#include <stdio.h>


int Option;
const int End  = 20;


// Bekanntmachung der Funktionen
void GnkWaendern();
void ListPerson();
int AddPerson();
long FindPerson();
long SendAnInt( short toSend );

void BildschirmAusgabeStart()
{
       cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";
       cout << "\t*****************************************************\n";
       cout << "\t*                                                   *\n";
       cout << "\t*              iScan Hauptmenue                     *\n";
       cout << "\t*                                                   *\n";
       cout << "\t*****************************************************\n";

       cout << "\n\n\n\t\t[1]\tIris erfassen\n\n";
       cout << "\t\t[2]\tEingetragene Personen anzeigen\n\n";
       cout << "\t\t[3]\tNeue Person eintragen\n\n";
       cout << "\t\t[4]\tGenauigkeit aendern\n\n";
       cout << "\t\t[5]\tProgramm beenden\n\n";
       cout << "\tBitte geben Sie die von Ihnen gewaehlte Option ein : \n\t" << flush;
       Option = getch();

}


int main()
{

       do
       {
              BildschirmAusgabeStart();

              // Testphase
              switch ( Option )
              {
              case '1':
                     // getIris();
                     FindPerson();
                     break;
```

```
            case '2':
                   ListPerson();
                   break;

            case '3':
                   AddPerson();
                   break;

            case '4':
                   GnkWaendern();
                   break;
            case '5':
                   //initialize COM
                   ::CoInitialize(NULL);

                   SendAnInt( (short)20 );

                   // uninitialize COM
                   ::CoUninitialize();


                   cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";
                   cout << "i S c a n    2 0 0 2\n\n";
                   cout << "DSP-Labor WS 01/02\n\n";

                   return 0;
                   break;
            default:
                   cout << "\tDie von Ihnen gewaehlte Option ist nicht bekannt!\n";
                   cout << "\tWeiter mit Return...\n" << flush;
                   Option = getch();
            }
       } while( Option != 5 ) ;

       return 0;
}


// Ende HostMain.cpp
```

Host.cpp: Diese Datei beinhaltet die RTDX Kommunikation

```
/************************************************************
*                                                                   *
*       RTDX                                                        *
*       DSP-Labor 2001/2002                                         *
*                                                                   *
*       Fazli Ayan                                                  *
*       Hakan Uluc                                                  *
*       Hikmet Citak                                                *
*                                                                   *
*       Last change: 10.02.2002                                     *
*                                                                   *
*************************************************************/


#import        "..\rtdxint.dll"
#include       <iostream.h>
#include       <conio.h>
using namespace RTDXINTLib;


// Deklaration der Konstanten
const long Success                  = 0;
const long Failure                  = 0x80004005;
const long ENoDataAvailable   = 0x8003001E;
const long EEndOfLogFile      = 0x80030002;

const short Datavolume        = 512;
const short Request                 = 5;
const short Acknowledge             = 10;
const short End                            = 20;


// Entwurf einer Klasse zur Beschreibung der Kanäle
```

```
class RTDX_Channel
{
public:
	IRtdxExpPtr rtdx;
	char *name;
	char *type;
	bool opened;
};


// Definition von drei Kanälen
RTDX_Channel rtdxArr[3];


// Deklaration der Variablen
short Command;			// holds data received from target
short Option;
long status;				// holds status of RTDX COM API calls
long bufferstate;
long i = 0;
HRESULT hr;				// holds status of generic COM API calls
VARIANT sa;				// holds pointer to the SAFEARRAY
long loopcounter;
unsigned char Buffer;
int j=0;




// Funktion, um eine ganze Zahl zu empfangen
long ReceiveAnInt ( short *toReceive , int Channelnum, char *Channelname)
{
	// Kanalname
	rtdxArr[Channelnum].name	= Channelname;

	// Kanal initialisieren
	rtdxArr[Channelnum].type		= "R";
	rtdxArr[Channelnum].opened	= false;

	hr = rtdxArr[Channelnum].rtdx.CreateInstance( L"RTDX" ); //Initiate the RTDX COM Object
	if ( FAILED(hr) )
	{
		cerr << hex << hr << " -	ERROR: Ininitiation ochan failed! \n";
		return -1;
	}

	status = rtdxArr[Channelnum].rtdx->Open(rtdxArr[Channelnum].name,
rtdxArr[Channelnum].type);
	if ( status != Success )
	{
		cerr << "set opened flag failed!!!";
		return Failure;
	}
	else
		rtdxArr[Channelnum].opened = true;

	// Lese Nachricht
	status = rtdxArr[Channelnum].rtdx->ReadI2( toReceive );
	if ( status != Success )
	{
		cerr << hex << status <<"	-Error:  cannot read Command! \n";
		return Failure;
	}
	rtdxArr[Channelnum].rtdx->Close();
	rtdxArr[Channelnum].opened = false;
	rtdxArr[Channelnum].rtdx.Release();
	return 0;
}


// Funktion, um eine ganze Zahl zu senden
long SendAnInt( short toSend )
{

	rtdxArr[1].name		= "ichan";
```

```
        // ichan initialisieren
        rtdxArr[1].type               = "W";
        rtdxArr[1].opened     = false;
        hr = rtdxArr[1].rtdx.CreateInstance( L"RTDX" ); //Initiate the RTDX COM Object
        if ( FAILED(hr) )
        {
                cerr << hex << hr << " -      ERROR: Ininitiation ichan failed! \n";
                return Failure;
        }


        status = rtdxArr[1].rtdx->Open(rtdxArr[1].name, rtdxArr[1].type);
        if ( status != Success )
        {
                cerr << hex << status <<"     -Error: Opening of channel "<<rtdxArr[1].name<<"
failure! \n";
                return -1;
        }

        else
                rtdxArr[1].opened = true;

        status = rtdxArr[1].rtdx->WriteI2( toSend, &bufferstate );
        status = rtdxArr[1].rtdx->Flush();
        if ( status != Success )
        {

                cerr << hex << status << " - Error: WriteI4 failed!\n";
                return -1;
        }


        status = rtdxArr[1].rtdx->Close();
        rtdxArr[1].rtdx.Release();

        return 0;
}


long getIris(unsigned char *matrix)
{
        //initialize COM
        ::CoInitialize(NULL);

        //initialize VARIANT
        ::VariantInit( &sa );

        // Kanalname
        rtdxArr[2].name       = "ochan_SAFEARRAY";

        // ochan_SAFEARRAY initialisieren
        rtdxArr[2].type               = "R";
        rtdxArr[2].opened     = false;
        hr = rtdxArr[2].rtdx.CreateInstance( __uuidof(RTDXINTLib::RtdxExp) ); //Initiate the
RTDX COM Object
        if ( FAILED(hr) )
        {
                //cerr << hex << hr << " -    ERROR: Ininitiation ochan failed! \n";
                return Failure;
        }

        // Kanal öffnen
        status = rtdxArr[2].rtdx->Open(rtdxArr[2].name, rtdxArr[2].type);

        if ( status != Success )
        {
                cerr << hex << status <<"     -Error: Opening of channel "<<rtdxArr[2].name<<"
failure! \n";
                return Failure;
        }
        else
        rtdxArr[2].opened = true;


        // REQ senden
        status = SendAnInt( Request );
```

```
if (status != Success)
{
        cerr << "\n\tSending failed!!!\n";
        return Failure;
}


// Antwort empfangen
status = ReceiveAnInt( &Command,0,"ochan");
if (status != Success)
{
        cerr << "\n\tReceiving failed!!!\n";
        return Failure;
}

if (Command == 10)
{
        cout<< "\n\t ----> T a r g e t   g e f u n d e n !\n"<<flush;
}
else
{
        cout<<"\n\tTarget nicht gefunden!\n"<<flush;
        return Failure;
}


/************************************************************
*                                                           *
*             Target ist gefunden Falls eine               *
*             fünf empfangen wurde                         *
*                                                           *
*************************************************************/
cout << "\n\tBitte druecken Sie eine Taste, um das Iris aufzunehmen...\n" << flush;
getch();                                    // Kommando für "Iris ist Bereit"

cout << "\tBitte warten ..." << flush;


// Request-Integer an die Targetapplikation senden
        status = SendAnInt( Command );
        if (status != Success)
        {
                cerr << "\n\tSending failed!!!\n";
                return Failure;
        }

        Sleep(2000); // Sleep(320000)


// iCode empfangen
do
{
        // iCode über ochan_SAFEARRAY lesen
        status = rtdxArr[2].rtdx->ReadSAI1( &sa );

    // stats auswerten
 switch ( status )
        {
        case Success:
                // Display data
                cout << "\n";

                // Getting the Identity Key of the packet
                for ( i = 0; i < (signed)sa.parray->rgsabound[0].cElements; i++)
                {
                        hr = ::SafeArrayGetElement( sa.parray, &i, &Buffer );
                        *(matrix+i) = Buffer;
                }
                break;

        case Failure:
                cerr << hex \
           << status \
           << " - Error: ReadSAI4 returned failure! \n";
     return Failure;
```

```
            case ENoDataAvailable:
                    cerr << "\n\nNo Data is currently available!\n";
                    return Failure;

            case EEndOfLogFile:
                    cout << "\n\n\tIrisdaten uebertragen!\n";
           break;

            default:
                    cerr << hex << status << " - Error: Unknown return code! \n";
           return Failure;
            }

       } while ( status != EEndOfLogFile );


       // Kanal deaktivieren
       status = rtdxArr[2].rtdx->Close();


       // Release the RTDX COM Object
       rtdxArr[2].rtdx.Release();


       // uninitialize COM
       ::CoUninitialize();

       // Clear Variant
       ::VariantClear( &sa );
       return 0;
}
```

Bank.cpp: Diese Datei beinhaltet Funktionsdefinitionen, vor allem für die Datenverwaltung

```
/************************************************************
*                                                                  *
*       Declaration of the functions                       *
*       DSP-Labor 2001/2002                                        *
*                                                                  *
*       Fazli Ayan                                                 *
*       Hakan Uluc                                                 *
*       Hikmet Citak                                               *
*                                                                  *
*       Last change: 10.02.2002                                    *
*                                                                  *
*************************************************************/


#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>
#include <conio.h>
#include <string.h>


// Deklaration der Konstanten
const short Request                  = 5;    // Request und Acknowledge um die
const short Acknowledge              = 10;   // Kommunikation zu überprüfen

const long Success                   = 0;
const long Failure                   = 0x80004005;
const long ENoDataAvailable   = 0x8003001E;
const long EEndOfLogFile      = 0x80030002;
const long Datavolume         = 512;


int GenauigkeitsWert = 90;                   // Default ist 90% Übereinstimmung
int Funcstatus;

void GnkWaendern();
long getIris(unsigned char *matrix);
long SendAnInt(int toSend);
```

```
int FehlerGrenzeCalc(int *GenauigkeitsWert);


class person
{
public:
        char Name[10];
        char Vorname[10];
    unsigned char IrisCode[Datavolume];


        person()
        {

                strcpy(Name,"");
                        strcpy(Vorname,"");
                        for (int i=0; i<512 ; i++)
                                IrisCode[i] = 0;

        }


        // Funktion, um eine neue Person aufzunehmen
        int getPersonal()
        {
                long status;
                cout << "\n\n\tPersonalien Eingabe";
                cout << "\n\t===================\n";
                cout<<"\n\n\tNachname\t: ";
                cin >> Name;
                cout << "\n\n\tVorname\t\t: ";
                cin >> Vorname;
                cout << "\n\n\tTarget wird gesucht..." << flush;

                status = getIris( IrisCode );
                        if ( status == Failure )
                                return 1;                               // Failure
                        else
                                return 0;
    }


        void showPersonal()
        {
                cout << "\n\n\tPersonalien von " << Name;
                cout << "\n\t=========================\n";
                cout<<"\n\n\tNachname\t:\t" << Name;
                cout << "\n\n\tVorname\t\t:\t" << Vorname;
        }

}; // Ende class person


// Funktion, um Iris Codes zu vergleichen
long CompIriscode(unsigned char *bank, unsigned char *trans , int FehlerGrenze)
{
        const unsigned char konst=1;
        int i,j;
        int count = 0;
        long Ergebnis[Datavolume];
        long Shifted;
        long LogBuffer;

        for ( j=0 ; j < Datavolume; j++)
        {
                Ergebnis[j] = bank[j] ^ trans[j];     //     ^ - EXOR
                for ( i=0 ; i < (sizeof(unsigned char)*8) ; i++ )
                {
                        Shifted = Ergebnis[j] >> i;
                        LogBuffer = konst & Shifted;
                        if ( LogBuffer != 0 )
                                count++;
                        if ( count  > FehlerGrenze )
                                return Failure;
                }
        }
```

```
        return count;
}



// Personen auflisten
void ListPerson()
{
        int counter=0;
        person Person;
        fstream File("Bank.dat", ios::in|ios::binary);
        if (!File)
        {
                cerr << "\nFehler bei der Dateioeffnung!!!";
                exit(1);
        }

        cout << "\n\n\tFolgende Personen sind im Datenbank vorhanden :\n";
        cout <<              "\t==============================================\n\n";
        while(!File.eof())
        {
                File.read((char*)&Person, sizeof(Person));
                if (!File.eof())
                {
                        cout << "\t"<<Person.Name;
                        counter++;
                }
                if (counter%6==0)
                        cout<<endl;
        }
        File.close();
        cout << "\n\n\t---------------------------------------------------\n";
        cout <<     "\tDie Datenbank enthaelt "<<counter<<" Personen.";
        cout << "\n\n\t\tWeiter mit einer Taste..."<<flush;
        getch();
}


// Person hinzufügen
int AddPerson()
{
        bool dubel= false;
        person VglPerson;
        person Person;
        fstream FileVgl("Bank.dat", ios::in|ios::out|ios::binary|ios::app);
        fstream File("Bank.dat", ios::in|ios::out|ios::binary|ios::app);

        Funcstatus = Person.getPersonal();
        if ( Funcstatus !=0 )
        {
                cerr << "\n\tKonflikt im Datenempfang!!!";
                cerr << "\n\tDaten werden ignoriert!";
                cerr << "\n\n\t\tweiter mit einer Taste...";
                getch();
                return 1;
        }
        while (!FileVgl.eof())
        {
                FileVgl.read((char*)&VglPerson, sizeof(VglPerson));
                if (!FileVgl.eof())
                {
                        if (!strcmp(Person.Name,VglPerson.Name))
                        {
                                cout << "\n\n\tDer angegebene Name existiert schon !";
                                cout << "\n\tBitte modifizieren Sie ihn.";
                                cout << "\n\t\tWeiter mit einer Taste..."<<flush;
                                getch();
                                File.seekg(0,ios::beg);
                                dubel = true;

                                break;
                        }
                }
        }
        FileVgl.close();
        if ( dubel == false )
```

```
        {
                File.write((char*)&Person, sizeof(Person));

                cout <<"\n\n\t"<< Person.Name<<" wurde erfolgreich eingelesen!\n";
                cout << "\n\t\tWeiter mit einer Taste..."<<flush;
                getch();
        }

File.close();
return 0;
}


// Person in der Datenbank suchen
long FindPerson()
{
        long Funcstatus;
        int ErrBitAnzahl ;
        int AltPerson;
        char AltPersonName[10];
        person VglPerson;
        person BnkPerson;

        AltPerson = FehlerGrenzeCalc(&GenauigkeitsWert);
        Funcstatus = getIris(VglPerson.IrisCode);
        if ( Funcstatus !=0 )
        {
                cerr << "\n\tKonflikt im Datenempfang!!!";
                cerr << "\n\tDaten werden ignoriert!";
                cerr << "\n\n\t\tweiter mit einer Taste..."<<flush;
                getch();
                return 1;
        }

        fstream File("Bank.dat",ios::in|ios::binary);
        while( !File.eof() )
        {
                //cout << "\n\tDaten werden gelesen!"<< flush;
                File.read((char*)&BnkPerson, sizeof(BnkPerson));
                ErrBitAnzahl = CompIriscode(BnkPerson.IrisCode,VglPerson.IrisCode,
FehlerGrenzeCalc(&GenauigkeitsWert));
                if  ( (ErrBitAnzahl != Failure) && ( ErrBitAnzahl < AltPerson ) )
                {
                        AltPerson = ErrBitAnzahl;
                        strcpy(AltPersonName,BnkPerson.Name);
                }
        }


        fstream FileR("Bank.dat",ios::in|ios::binary);
        while ( !FileR.eof() )
        {
                FileR.read((char*)&BnkPerson, sizeof(BnkPerson));
                if(!strcmp(BnkPerson.Name, AltPersonName))
                {
                        BnkPerson.showPersonal();
                        cout << "\n\n\tDiese Person wurde mit einer Genauigkeit von
"<<GenauigkeitsWert<<" % ermittelt.";
                        cout << "\n\n\t\tWeiter mit einer Taste..."<<flush;
                        getch();
                        break;
                }
                if ( FileR.eof() )
                {
                        cout << "\n\n\n\tKeine Dateiuebereinstimmung gefunden!";
                        cout << "\n\n\t\tweiter mit einer Taste..."<<flush;
                        getch();
                        break;
                }
        }
        FileR.close();
        return 0;
}


// Genauigkeitswert berechnen
```

```
int FehlerGrenzeCalc(int *GenauigkeitsWert)
{
	int GroesseInBit;
	int Ergebnis;

	// Das Runden fehlt hier noch!
	GroesseInBit = Datavolume * sizeof(unsigned char)*8;
	Ergebnis = ((100 - *GenauigkeitsWert ) * GroesseInBit) / 100;
	return (Ergebnis);
}


// Genauigkeitswert ändern
void GnkWaendern()
{
	bool EndOfLoop;
	do
	{
		cout << "\n\n\tGenauigkeit aendern:";
		cout <<          "\n\t===================\n";
		cout << "\n\tDie Genauigkeit liegt zur Zeit bei "<< GenauigkeitsWert<< " %";
		cout << "\n\n\tBitte geben Sie einen neuen Genauigkeitswert ein [0-100%]:
"<<flush;
		cin >> GenauigkeitsWert;
		if ( GenauigkeitsWert > 100 )
		{
			cout << "\n\n\tDer von Ihnen angegebener Wert überschreitet 100\n";
			cout << "\tund wird deshalb ignoriert!\n";
			cout << "\n\t\tweiter mit einer Taste..."<<flush;
			getch();
			EndOfLoop = false;
		}
		else
		{
			cout << "\n\n\tDer Genauigkeitswert beträgt jetzt "<<
GenauigkeitsWert<<" %";
			cout << "\n\n\t\tweiter mit einer Taste..."<<flush;
			getch();
			EndOfLoop = true;
		}
	}while ( EndOfLoop == false );
}
```

7 Referenzen

(1) Die Simulationen der ersten, verworfenen Ansätze, sowie die der ersten Version des Drei-Punkte-Algorithmus sind in der beigefügten CD (Ordner *matlab*) enthalten. Siehe die Datei *matlab.txt.*

(2) Homepage von John Daughman: http://www.cl.cam.ac.uk/users/jgd1000/

(3) Anil K. Jain: Fundamentals Of Digital Image Processing (Prentice Hall, 1989)

(4) Texas Instruments: TMS320C62X Image/Video Library Programmer's Reference (Datei *spru400.pdf* im CodeComposer Studio).

(5) Brian K. Kernighan, Dennis M. Ritchie: The C Programming Language (Prentice Hall, 1991)