Elektronische Messdatenverarbeitung

Klaus Betzler *

Universität Osnabrück

Wintersemester 2006/2007

Inhaltsverzeichnis

* Klaus.Betzler@Physik.Uni-Osnabrueck.de 14. Oktober 2006

1 Detektoren und Sensoren

Experimentelle Messgrößen liegen im Regelfall nicht in 'EDV-kompatibler' Form vor. Diese herzustellen, d. h. die physikalische Größe in eine geeignete elektrische umzuwandeln – geeignet letztlich zur Weiterverarbeitung mit einem *Interface* und einem Computer – ist die Aufgabe von Detektoren und Sensoren. Die Abgrenzung zwischen den beiden Begriffen ist nicht immer eindeutig; will man abgrenzen, so kann man den Begriff Detektor für den Nachweis von Teilchen (Elektronen, Photonen), den Begriff Sensor für die Wandlung anderer physikalischer Größen (Temperatur, Lichtintensität) benutzen.

Bei der Anwendung eines Detektors oder Sensors sollte man sich in jedem Fall zunächst die zugrunde liegende physikalische Wirkungsweise klar machen, daher:

> Behandeln sie einen Detektor oder Sensor erst dann als *black box*,
> wenn sie wirklich wissen, was drin ist.
>
> Wichtig sind die Fähigkeiten *und* die Grenzen eines Systems.

Das gesamte Gebiet ist sicherlich zu groß, um es in einer Vorlesung auch nur annähernd vollständig behandeln zu können, wir beschränken uns daher auf einige Beispiele aus typischen Bereichen.

1.1 Temperatur

Praktisch alle physikalischen Eigenschaften sind mehr oder weniger stark temperaturabhängig, können also prinzipiell zur Temperaturmessung eingesetzt werden; anschaulichstes Beispiel ist die thermische Ausdehnung bei Festkörpern, Flüssigkeiten oder Gasen. Für die Anwendung in Sensoren besonders interessant und fast ausschließlich verwendet sind temperaturabhängige Änderungen der *elektrischen* Eigenschaften, die auf einfache Weise automatisch detektiert und elektrisch weiterverarbeitet werden können.

1.1.1 Widerstände

Speziell zur Temperaturmessung und -regelung entwickelte PTC- und NTC-Thermistoren, d. h. Widerstände mit positivem oder negativem Temperaturkoeffizienten (Kaltleiter, Heißleiter), fanden und finden in der technischen Elektrik breite Verwendung. Ihr Anwendungsbereich ist allerdings meist auf Temperaturen beschränkt, in denen technische Geräte, Haushaltsgeräte etc. arbeiten.

In einem weiteren Temperaturbereich einsetzbar und daher für physikalische Experimente interessanter (aber auch teurer) sind Platin-Widerstände mit besonderen Spezifikationen (Pt 100), die eine sehr ausgeprägte, gut definierte und dokumentierte Temperaturabhängigkeit zwischen etwa 10 K und 1000 K aufweisen.

Für tiefe Temperaturen (1 K bis 100 K) geeignet sind Kohle-Widerstände und spezielle Halbleiter-Widerstände (Ge) oder -Dioden (Si), die man auch mit genauer, individuell erstellter Eichung (dann sehr teuer) kaufen kann.

Bei der Temperaturmessung mit Widerständen ist – insbesondere bei tiefen Temperaturen – darauf zu achten, dass die zur Messung benötigte Leistung und damit die Wärmezufuhr möglichst gering ist.

1.1.2 Thermoelemente

Thermoelemente nutzen die materialspezifische Temperaturabhängigkeit der Ladungsträgerverteilung in Metallen oder Halbleitern aus (detaillierte Beschreibung in Lehrbüchern zur Festkörperphysik [1] oder Halbleiterphysik [2]). Obwohl der thermoelektrische Effekt (Thermokraft) im allgemeinen in Halbleitern größer ist, werden aus naheliegenden praktischen Gründen geeignete Kombinationen aus verschiedenen Metallen oder Metall-Legierungen verwendet. Einige gebräuchliche *Thermopaare* sind in Tabelle 1 zusammengestellt.

Thermopaar	Temperaturbereich	Diff. Thermospannung bei 0 °C
Gold Eisen – Nickel Chrom	-270 °C... 0 °C	20 μV/K
Kupfer – Konstantan	-200 °C... 600 °C	40 μV/K
Eisen – Konstantan	-200 °C... 900 °C	52 μV/K
Nickel Chrom – Konstantan	0 °C... 1000 °C	63 μV/K
Nickel Chrom – Nickel	-200 °C... 1370 °C	40 μV/K
Platin Rhodium – Platin	0 °C... 1750 °C	55 μV/K
Wolfram Rhenium 5 – 26	0 °C... 2500 °C	10 μV/K

Tabelle 1: Typischer Anwendungstemperaturbereich und differentielle Thermospannung gebräuchlicher Thermoelemente.

Die Vorteile von Thermoelementen liegen in ihrem großen Anwendungstemperaturbereich, ihrer relativ einfachen Handhabung und dem günstigen Preis. Nachteilig ist das geringe elektrische Signal, das ein empfindliches Messgerät bzw. eine stabile Verstärkung erfordert, und die Notwendigkeit einer Referenzstelle (Eisbad) mit definierter, konstanter Temperatur (andere Möglichkeit s. 1.1.3).

Bei der Herstellung von Thermoelementen ist insbesondere auf guten elektrischen Kontakt zwischen den beiden Materialien zu achten (Punkt- oder Mikroschweißung). Teure Thermoelemente können mit *Ausgleichsleitungen* angeschlossen werden. Genauer ist dies in den Datenblättern der Hersteller beschrieben, die meist auch ausführliche Tabellen der temperaturabhängigen Thermospannungen enthalten.

1.1.3 Band-Gap-Referenz-Diode

Die Strom-Spannungs-Kennlinie einer Halbleiterdiode wird in guter Näherung durch

$$I = I_S(\exp \frac{eU}{kT} - 1) \tag{1.1}$$

beschrieben. Dies wurde schon in der Frühzeit der Halbleiterphysik von Shockley hergeleitet und experimentell verifiziert [3].

Abbildung 1: Strom-Spannungs-Kennlinien einer Diode nach Gleichung 1.1 für zwei verschiedene Temperaturen (10 °C und 50 °C); der Strom im Sperrbereich ist um 10^4 überhöht dargestellt.

Im Durchlassbereich ($U \gg kT/e$) kann man die -1 in Gl. 1.1 gegen die Exponentialfunktion vernachlässigen, bei konstantem Strom I ist dann der Zusammenhang zwischen Spannung U und Temperatur T annähernd linear (die Temperaturabhängigkeit des Sättigungssperrstroms I_S ist bei dieser Betrachtung nicht berücksichtigt). Basierend auf diesem Effekt lassen sich Temperaturfühler mit sehr gut linearer Kennlinie bauen [4]. Ein typisches Beispiel ist der integrierte Schaltkreis AD 592 von Analog Devices.

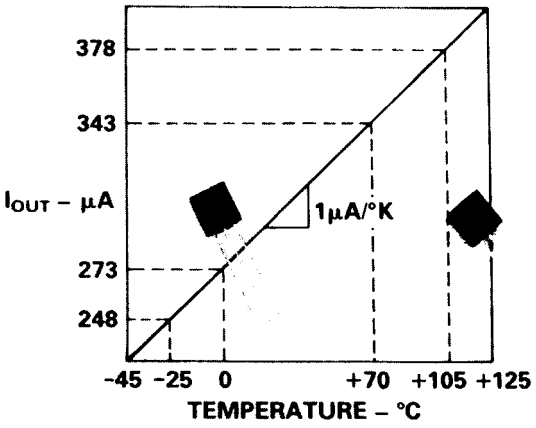


Abbildung 2: Idealisierte Kennlinie und Bauform des IC AD 592; aus [5].

Der AD 592 liefert in einem Betriebsspannungsbereich 3 V... 30 V einen sehr gut temperaturproportionalen Strom, der Proportionalitätsfaktor ist genau 1 μA/K (Abb. 2), dies im Temperaturbereich -25 °C... 105 °C. An einem Lastwiderstand von beispielsweise 10 kΩ ergibt dies eine gut messbare Spannung von einigen Volt. Die Nichtlinearität der Kennlinie liegt – auf die Temperatur umgerechnet – in der Größenordnung von $\pm$0.5 K. Die Bauform (Plastikgehäuse) bedingt allerdings eine relativ große thermische Zeitkonstante und schlechte Wärmeabfuhr, die diesbezüglichen Daten sind in Tabelle 2 zusammengestellt.

Bevorzugte Anwendungsgebiete sind mithin solche mit langsam veränderlicher Temperatur, beispielsweise die Temperaturregelung von Halbleiterlasern oder die Messung der

Art der Kühlung	Wärmewiderstand	Therm. Zeitkonstante
Ruhende Luft	175 K/Watt	60 sec
Ruhende Luft + Kühlblech	130 K/Watt	55 sec
Bewegte Luft	60 K/Watt	12 sec
Bewegte Luft + Kühlblech	40 K/Watt	10 sec
Flüssigkeit	35 K/Watt	5 sec
Aluminiumblock + Wärmeleitpaste	30 K/Watt	3 sec

Tabelle 2: Wärmewiderstand und thermische Zeitkonstante des AD 592 für verschiedene Anwendungsarten. Die Erwärmung durch die im Schaltkreis umgesetzte Leistung kann demnach im ungünstigsten Fall bis zu 2 K betragen.

Referenztemperatur bei Thermoelementen. Einen Vorschlag zur Referenzstellenkompensation zeigt Abbildung 3.

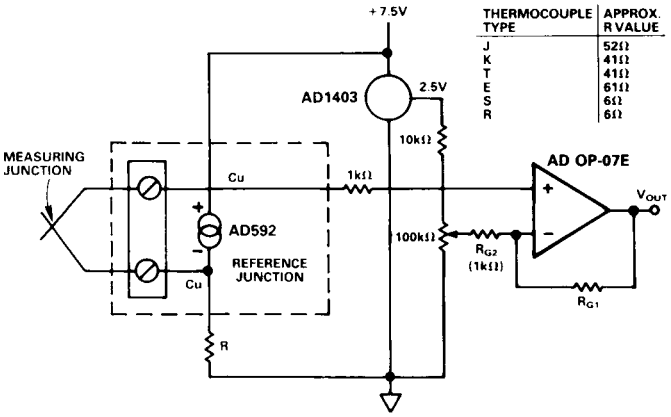


Abbildung 3: Referenzstellenkompensation mit dem IC AD 592 bei der Temperaturmessung mit einem Thermoelement (Schaltungsvorschlag aus dem Datenblatt des IC-Herstellers Analog Devices); aus [5].

1.1.4 Kapazitive Sensoren

Die bisher beschriebenen Temperatursensoren werden ungenau, wenn große Magnetfelder am Messort vorhanden sind. Für diesen Spezialfall können kapazitive Sensoren verwendet werden, die als physikalisches Messprinzip die Abhängigkeit der Dielektrizitätskonstanten von der Temperatur ausnutzen (die Kapazitätsmessung ist allerdings deutlich aufwendiger als etwa eine Widerstandsmessung). Besonders geeignet dafür sind Materialien, die in der Nähe der zu messenden Temperaturen einem strukturellen Phasenübergang (paraelektrisch → ferroelektrisch) zustreben. Ein Beispiel ist Strontiumtitanat bei sehr tiefen Temperaturen, die Dielektrizitätskonstante nimmt gegen 0 K deutlich zu, ohne dass ein Phasenübergang tatsächlich erreicht wird.

1.2 Koordinaten (Ort und Winkel)

Die klassischen Sensoren für diesen Bereich sind Potentiometer, lineare oder Drehwiderstände, die – direkt oder über mehr oder weniger aufwendige Getriebe mit der Messstelle verbunden, mit konstantem Strom oder konstanter Spannung betrieben – eine orts- oder winkelabhängige Spannung liefern. Soll's genauer sein, kann man bei kleinen Wegen kapazitive oder piezoelektrische Prinzipien verwenden, bei größeren optische Phasenmessungen (Interferometer), bei sehr großen optische oder elektrische Laufzeitmessungen.

Im Werkzeugmaschinenbereich werden derzeit hauptsächlich exakte mechanische Maßstäbe oder Teilscheiben verwendet, die optoelektronisch abgelesen werden. Auf Spezialglassubstrate werden hochgenaue Teilungen oder Kodierungen aufgedampft, Beispiele für Teilscheiben zur inkrementellen oder absoluten Winkelmessung zeigt Abbildung 4.

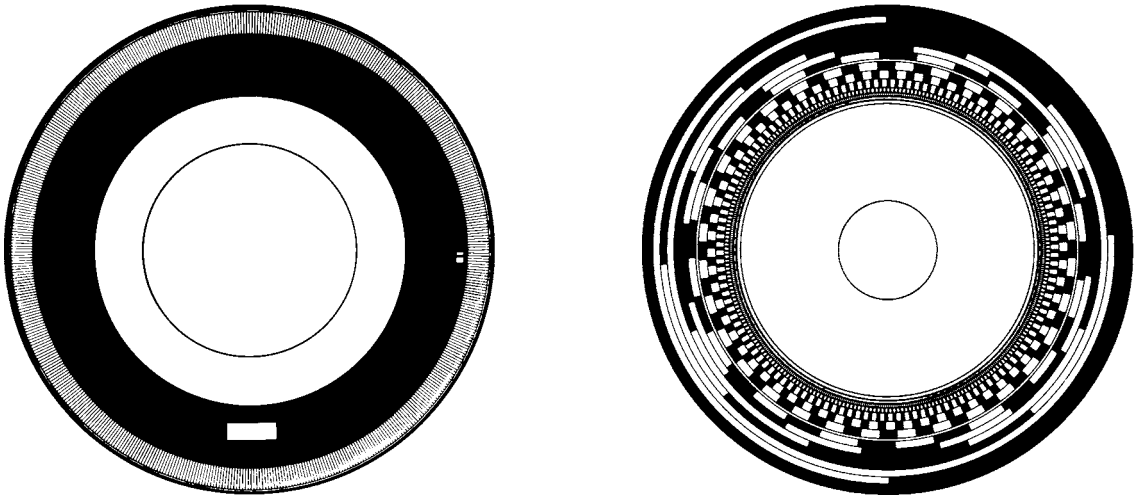


Abbildung 4: Teilscheibe eines inkrementellen Drehgebers (links) und eines Code-Drehgebers (rechts), beide etwa in natürlicher Größe (entnommen einem Katalog der Firma Heidenhain, Traunreut). Die Code-Teilscheibe rechts ist im Gray-Code kodiert, einem binären Code, bei dem sich von einem Wert zum nächsten immer nur ein Bit der Kodierung verändert; auf diese Weise können keine Ablesefehler auftreten, wenn die Scheibe zwischen zwei Werten steht.

Die mit solchen Teilscheiben aufgebauten inkrementellen Drehgeber liefern in der Regel zwei um $\pi/2$ gegeneinander phasenverschobene Sinusspannungen, die in der zugehörigen Anzeigeelektronik ausgewertet werden. Die aktuelle Winkelposition kann über eine Standardschnittstelle (Seriell oder parallel) in einen angeschlossenen Rechner übernommen werden. Benötigt man die Messpunkte in schneller Abfolge, bietet es sich an, die Sinussignale über Analog-Digital-Wandler direkt vom Rechner zu erfassen und auszuwerten. Die mechanische Genauigkeit der Teilscheiben und der optoelektronischen Ablesung ist so gut, dass pro Sinusperiode mehrere hundert Subschritte interpoliert werden können. Damit kann eine Genauigkeit der Winkelmessung erzielt werden, die in der Größenordnung von 1/1000 Grad liegt.

1.3 Licht

Zum Nachweis von Licht können unterschiedliche physikalische Effekte ausgenutzt werden. Einerseits sind dies Quanteneffekte (äußerer oder innerer Photoeffekt), mit denen sehr empfindliche Detektoren realisiert werden können, andererseits Effekte, die in irgendeiner Form die Temperaturänderung nachweisen, die durch den mit Licht verbundenen makroskopischen Energiestrom bewirkt wird (eine ausführliche Beschreibung der bei Photodetektoren genutzten physikalischen Effekte und der damit realisierten Bauelemente gibt beispielsweise [6]). Wegen der wesentlich besseren Empfindlichkeit sind für physikalische Experimente in erster Linie die auf den Photoeffekten basierenden Detektoren interessant. Thermische Detektoren werden generell nur dort eingesetzt, wo entweder sehr hohe Lichtleistungen zu messen sind oder wo in Wellenlängenbereichen gemessen werden muss, die für andere Detektoren nicht zugänglich sind.

1.3.1 Zur Charakterisierung von Photodetektoren

Innerer und äußerer Photoeffekt sind als 'Bandstruktureffekte' sehr stark material- und energieabhängig. Ladungsträger können – sieht man von in der Regel vernachlässigbaren Mehrquanteneffekten ab – nur von Photonen ab einer bestimmten Quantenenergie angeregt werden. Die Nachweisempfindlichkeit ist auch oberhalb dieser Grenzenergie stark wellenlängenabhängig. Bei thermische Detektoren ist im Vergleich dazu der Empfindlichkeitsverlauf wesentlich weniger dramatisch. Zur Charakterisierung von Detektoren wird die Empfindlichkeit mit ihrer Wellenlängenabhängigkeit angegeben – als *Responsivity* $R(\lambda)$, gemessen beispielsweise in Ampere/Watt, bei Photodioden oder als Quantenausbeute, gemessen in %, bei Kathoden von Photomultipliern.

***NEP*-Wert:** Bei geringen Lichtsignalen stellt das Detektorrauschen eine ganz wesentliche Begrenzung dar, es ist daher üblich, dieses Rauschen bei der Empfindlichkeitsangabe implizit zu berücksichtigen. Als Maß für die so definierte Empfindlichkeit wird die Lichtleistung angegeben, die notwendig ist, um ein dem Rauschen äquivalentes Ausgangssignal zu generieren (*NEP*-Wert – *N*oise *E*quivalent *P*ower).

Detektivität: Detektoren sind umso besser, je kleiner ihr *NEP*-Wert ist. Da große Werte jedoch allgemein beliebter und vor allem werbewirksamer sind, wird häufig die dazu reziproke Größe, die Detektivität $D = 1/NEP$ verwendet.

Spezifische Detektivität: Das Detektorrauschen hängt nicht nur von der Detektorart ab, sondern auch von der Detektorgröße sowie von den Messbedingungen (Detektortemperatur, Verstärkerbandbreite). Weiterhin kann der Erfassungswinkel von Detektoren unterschiedlich sein. Als statistische Größe ist das Rauschsignal in guter Näherung proportional zur Quadratwurzel aus der Detektorfläche A und der Bandbreite B. Man objektiviert daher üblicherweise von den genannten Nebenbedingungen durch Angabe der spezifischen Detektivität D^* (*D-Stern*):

$$D^* = (NEP)^{-1} \cdot A^{1/2} \cdot B^{1/2} \,. \tag{1.2}$$

Einige Autoren bzw. Firmen definieren zusätzlich ein *D-Doppelstern* $D^{**} = D^* \sin\Theta$, das zusätzlich den Erfassungswinkel (Θ = halber Erfassungswinkel) berücksichtigt.

Einen Überblick der Detektivitätskurven verschiedener Detektortypen geben die Abbildungen 5 und 6, Abbildung 5 für Anwendungen im sichtbaren Spektralbereich sowie im nahen Ultraviolett und Infrarot (Photomultiplier und -dioden), Abbildung 6 für den Infrarotbereich zwischen 1 und 1000 μm (Photoleiter und thermische Detektoren).

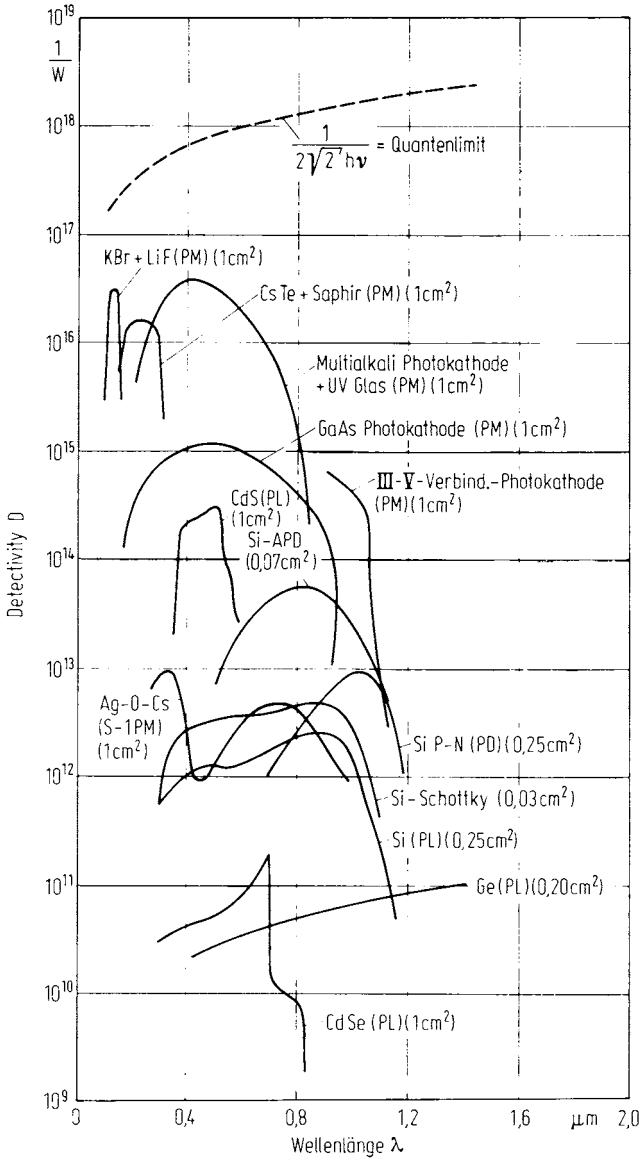


Abbildung 5: Detektivität D von Photomultipliern (PM), Photoleitern (PL), Photodioden (PD) und Lawinenphotodioden (APD); aus [6].

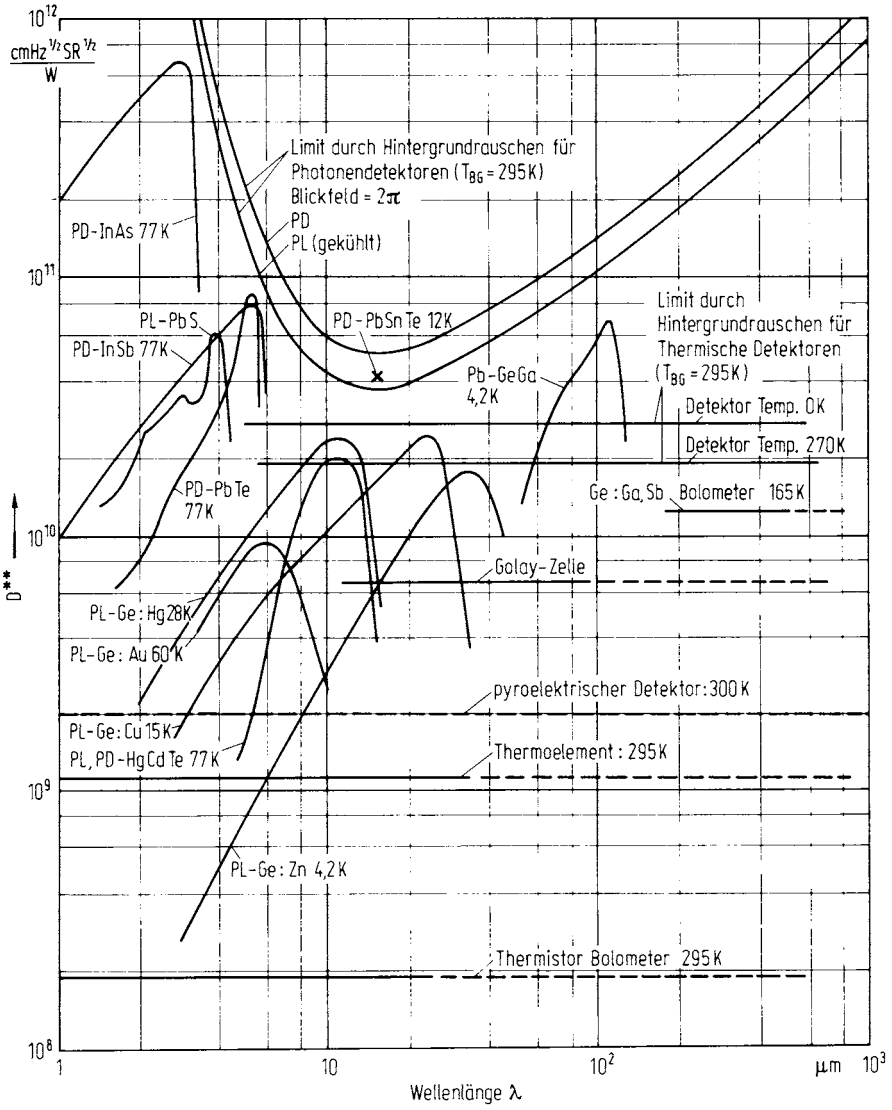


Abbildung 6: Spezifische Detektivität D^{**} von thermischen Detektoren und Photonendetektoren; aus [6].

Rauschen und Energiebereich: Während bei ausreichenden Lichtintensitäten das Detektorrauschen keine große Rolle spielt, ist es bei der Messung geringer Intensitäten die begrenzende Größe für die minimal messbare Intensität. Bei Photonendetektoren wird das Rauschen bei geringen Lichtintensitäten (Dunkelrauschen) im wesentlichen durch die thermische Anregung von Ladungsträgern verursacht. Die Aktivierungsenergie für diese thermische Anregung ist generell kleiner als die für optische, daher ist das thermisch generierte Detektorrauschen umso größer, je geringer die optische Anregungsenergie ist, d. h. je langwelliger ein Detektor nutzbar ist. Durch Kühlung des Detektors lassen sich

die Verhältnisse zwar etwas verbessern, man wird im allgemeinen jedoch immer einen für die jeweilige Messaufgabe optimierten Kompromiss schließen müssen. Für geringe Intensitäten sieht der oft so aus, dass man einen Detektor verwendet, mit dem der gewünschte Wellenlängenbereich gerade noch abzudecken ist.

1.3.2 Photomultiplier

Zum Nachweis extrem geringer Lichtintensitäten im sichtbaren, ultravioletten und nahinfraroten Spektralbereich verwendet man meist Photomultiplier. Ihre Funktionsweise beruht auf dem äußeren Photoeffekt und der Sekundärelektronenemission: aus einer Photokathode (im Vakuum) werden durch Lichtquanten Elektronen ausgelöst, die durch eine Spannung von etwa 100 V zur nächsten Elektrode (Dynode) beschleunigt werden, wo jedes mehrere Sekundärelektronen auslöst. Solche Verstärkungsstufen werden kaskadiert (ca. 10), so dass an der letzten Elektrode (Anode) ein gut messbares Ladungssignal entsteht (Sekundärelektronen-Vervielfacher). Mit dafür optimierten Anordnungen können so problemlos einzelne Photonen nachgewiesen werden (*photon counting*). Den prinzipiellen Aufbau eines Photomultipliers zeigt Abbildung 7.

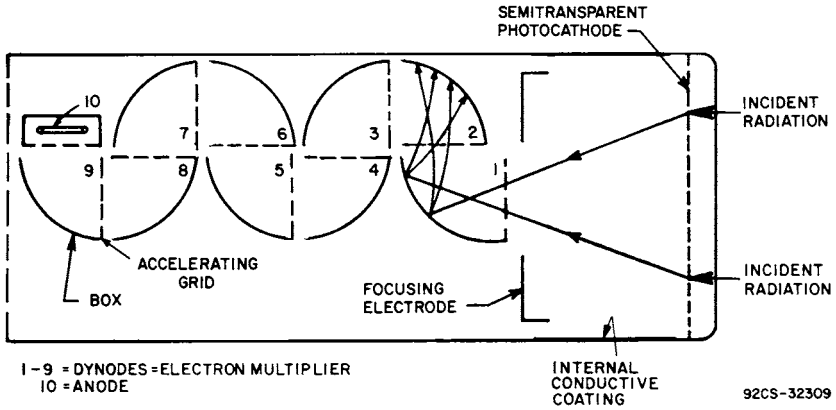


Abbildung 7: Typische Geometrie eines Photomultipliers mit Frontfensterkathode (gebräuchliche Durchmesser liegen zwischen 10 und 100 mm); aus [7].

Kathodenmaterialien: Für Photokathoden werden Materialien aus drei Substanzklassen verwendet: Metalle mit niedriger Austrittsarbeit (meist Mischungen aus Alkalimetallen) für den sichtbaren Spektralbereich, Halbleiter (Telluride, Oxide) für das Ultraviolette und III-V-Halbleiter (GaAs, GaInAs), bei denen mit geeigneter Beschichtung eine *negative Elektronenaffinität* erreicht wurde, für das nahe Infrarot. Die typischen Bandstrukturen (Energienivaues im Ortsraum) sind in Abbildung 8 skizziert.

Die Kathoden sind entweder als dünne semitransparente Schicht innen auf das Frontfenster aufgedampft oder als massivere Beschichtung auf ein Metallblech aufgetragen. Die Kathodengrößen liegen für optische Anwendungen zwischen einigen Millimetern (rauscharme Photonenzählanwendungen) und einigen Zentimetern.

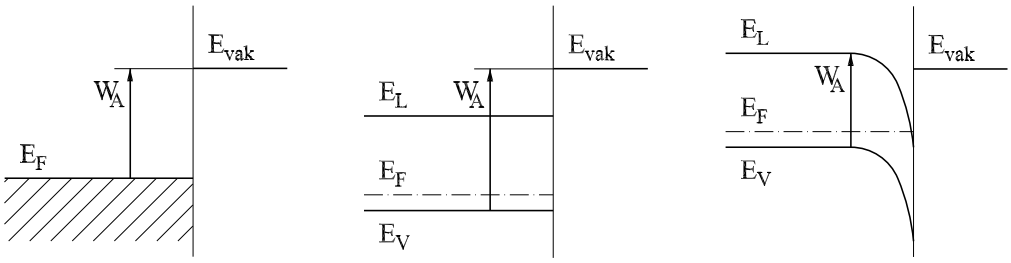


Abbildung 8: 'Optische' Elektronenaustrittsarbeit W_A bei Metallen (linkes Bild), Halbleitern (mittleres Bild) und Halbleitern mit negativer Elektronenaffinität, *d. h. $E_{vak} < E_L$ (rechtes Bild). E_F: Fermi-Energie, E_V: Valenz-, E_L: Leitungsband, E_{vak}: Vakuumniveau.*

Empfindlichkeit: Das Maximum der Quantenausbeute bei Photomultipliern liegt je nach Kathodenmaterial zwischen 0.1 und 30 %, d. h. jedes tausendste bzw. dritte auf die Photokathode treffende Photon löst dort ein Elektron aus. Der spektrale Verlauf ist im langwelligen Bereich durch die Austrittsarbeit bestimmt, im kurzwelligen Bereich in der Regel durch das Fenstermaterial[1]. Eine Übersicht über die spektralen Empfindlichkeitskurven verschiedener Kathodenmaterialien[2] zeigen die Abbildungen 9, 10 und 11 (aus [8]).

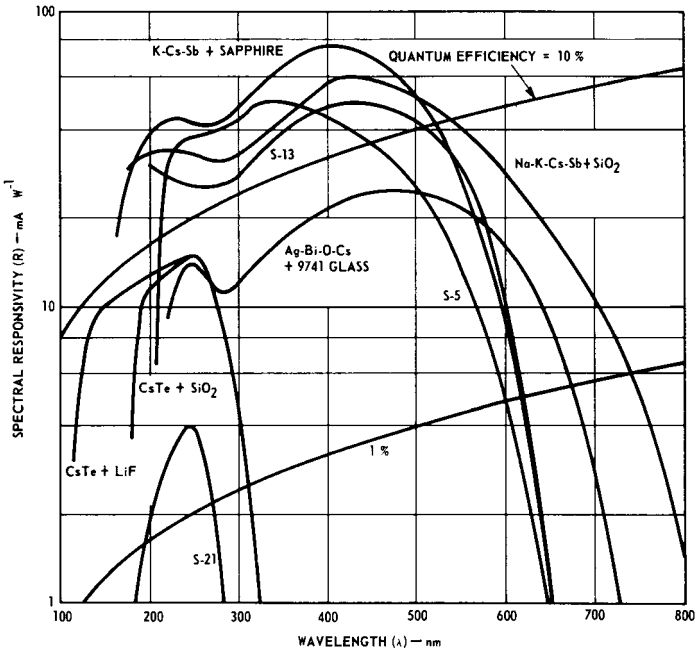


Abbildung 9: Empfindlichkeitskurven von Photomultipliern für Ultraviolettanwendungen.

[1] Empfindlichkeit im ferneren UV kann auch durch Szintillatormaterialien erreicht werden, die kurzwelliges Licht absorbieren und längerwellig lumineszieren.

[2] Die teilweise verwendeten Bezeichnungen S4, S11 usw. sind gebräuchliche Trivialnamen für bestimmte Materialmischungen.

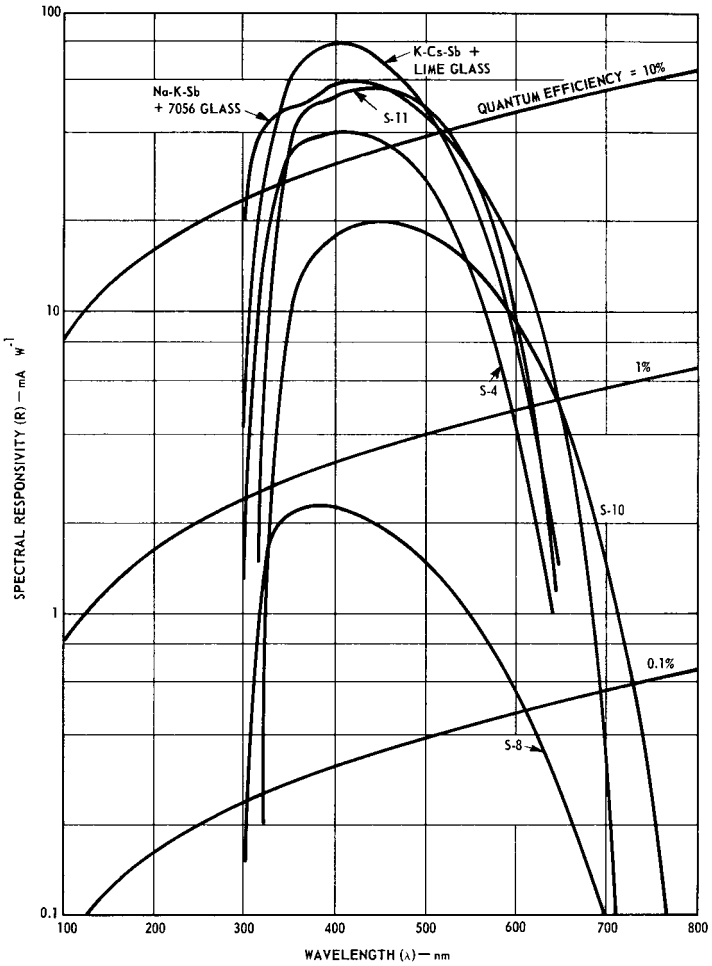


Abbildung 10: Empfindlichkeitskurven von Photomultipliern für Anwendungen im sichtbaren Spektralbereich.

Dynoden werden aus Materialien mit guter Sekundärelektroneneffizienz hergestellt, vorwiegend aus BeO (gute Hochtemperatureigenschaften) oder Cs_3Sb.

Zeitverhalten: Die *Gesamtlaufzeit* der Elektronen von der Kathode zur Anode beträgt je nach Bauform zwischen 10 und 100 nsec. Durch eine Optimierung der Dynodenanordnung kann man erreichen, dass die *Laufzeitstreuung*, damit die Verbreiterung eines Pulses, deutlich unter diesen Zeiten bleibt. Erreichbar sind Werte in der Größenordnung einer Nanosekunde. Dies begrenzt grundsätzlich die mit Photomultipliern erreichbare Zeitauflösung, aber auch die für Photonenzählanwendungen maximal mögliche Zählrate.

Verstärkung: Typische Verstärkungen der verwendeten Dynodenanordnungen liegen zwischen 10^4 und 10^7. Dass dies ausreicht, zeigt eine Abschätzung des für ein einzelnes nach-

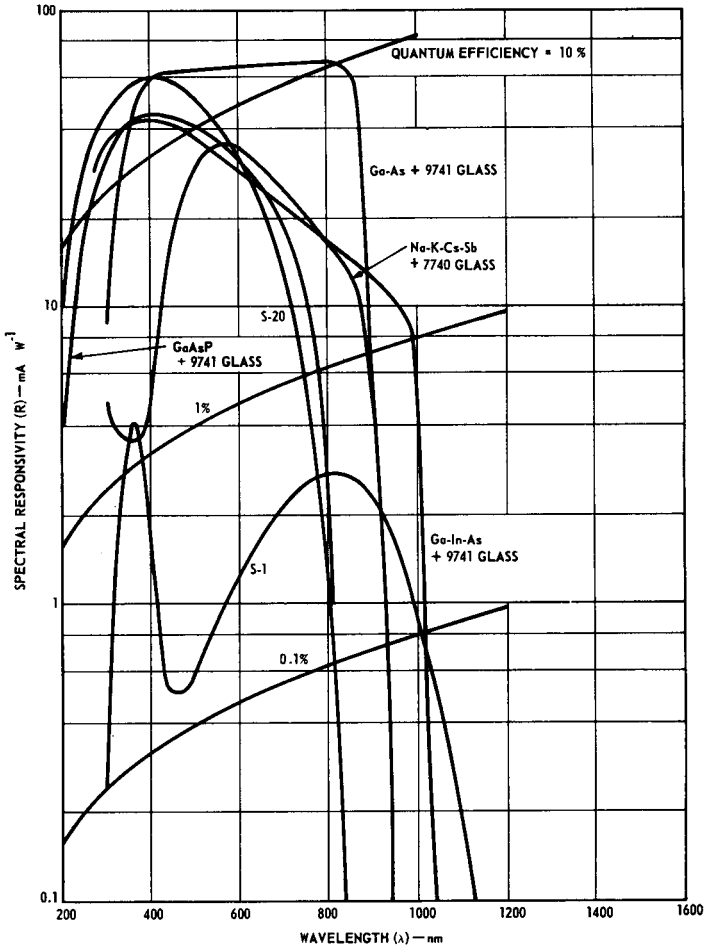


Abbildung 11: Empfindlichkeitskurven von rot- und infrarotempfindlichen Photomultipliern.

gewiesenes Photon an der Anode entstehenden Spannungsimpulses

$$U_{\text{puls}} = e \cdot V \cdot R \cdot \tau^{-1} \ . \tag{1.3}$$

An einem Lastwiderstand $R = 75\,\Omega$ werden bei einer Verstärkung $V = 10^7$ und einer Laufzeitstreuung $\tau = 10^{-9}\,\text{sec}$ etwa 100 mV erreicht.

Versorgungsspannung: Die Verstärkung einer einzelnen Dynodenstrecke ist leicht sublinear von der Beschleunigungsspannung abhängig, $V_1 \propto U_1^x$ mit $x = 0.7 \ldots 0.9$, bedingt durch die bei höheren Energien größere Eindringtiefe. Bei n Dynoden ist die Gesamtverstärkung

$$V \propto (U_1^x)^n = U_1^{nx} \propto U_{\text{B}}^{nx} \ , \tag{1.4}$$

mithin sehr empfindlich von der Betriebsspannung U_{B} abhängig. Wegen dieser Abhängigkeit muss die Versorgungsspannung sehr gut konstant und störungsfrei gehalten werden.

Die einzelnen Dynodenspannungen werden durch einen hochohmigen Spannungsteiler eingestellt (meist im Gehäuse direkt am Sockel aufgebaut). Der Spannungsteiler soll einerseits hochohmig sein, um wenig Verlustwärme zu produzieren, andererseits niederohmig genug, um zu gewährleisten, dass sich bei den im Betrieb auftretenden Lichtintensitäten die Spannungsverhältnisse an den Dynoden nicht merklich verändern. An der Kathode liegt die (negative) Betriebsspannung an, der Anodenstrom wird zur Betriebsspannungsmasse hin gemessen[3].

Mikrokanalplatten: Eine Sonderform der Sekundärelektronenvervielfacher sind Mikrokanalplatten. Die Vervielfachung findet dort nicht diskretisiert auf Dynoden sondern quasikontinuierlich in geeignet beschichteten 'Kanälen' statt. Vorteile der Mikrokanalplatten

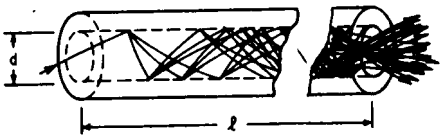


Abbildung 12: Sekundärelektronenvervielfachung in einem 'Kanal' einer Mikrokanalplatte.

sind kompakte Bauform, Unempfindlichkeit gegen Magnetfelder und die Möglichkeit einer ortsaufgelösten Vervielfachung (Bildverstärker), Nachteil ist der durch den großen Herstellungsaufwand bedingte hohe Preis.

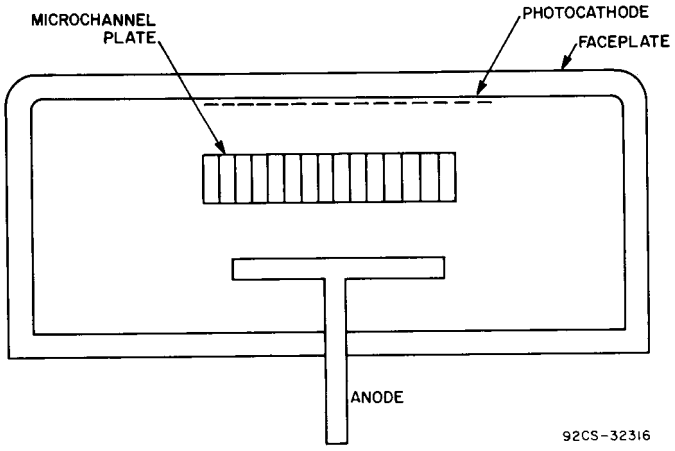


Abbildung 13: Photomultiplier mit einer Mikrokanalplatte als Sekundärelektronenvervielfacher; aus [7].

[3] Bei Impulsanwendungen (Szintillationszähler) macht man's im allgemeinen umgekehrt, die Kathode liegt auf Masse (größere Betriebssicherheit), der Ladungsimpuls an der auf Hochspannung liegenden Anode wird durch einen Kondensator ausgekoppelt. Fast alle Hochspannungsversorgungsgeräte für Photomultiplier können zwischen diesen beiden Betriebsarten umgeschaltet werden.

1.3.3 Photodioden, Phototransistoren

Die Funktionsweise von Photodioden und Phototransistoren ist in Lehrbüchern zur Halbleiterphysik meist recht ausführlich beschrieben [9]. Die Skizzenfolge der nebenstehenden Abbildung 14 soll die grundlegenden Mechanismen verdeutlichen, skizziert sind die Energieverhältnisse im Ortsraum (Bandstruktur).

p-n-Übergang im Gleichgewicht: Bringt man (was so natürlich nur theoretisch möglich ist) einen p- und einen n-dotierten Halbleiter miteinander in Kontakt, so stellt man dadurch einen 'abrupten' p-n-Übergang her. Die Fermi-Energie E_F bzw. das chemische Potential liegt beim p-Halbleiter in der Nähe des Valenzbandes (E_V), beim n-Halbleiter in der Nähe des Leitungsbandes (E_L) – Teilbild (a). Beim Kontakt fließen solange bewegliche Ladungen (Elektronen und Löcher) aus dem kontaktnahen Bereich ab, bis die Fermi-Energie im gesamten System konstant ist (b). Die ionisierten Akzeptoren (Dichte N_A) und Donatoren (Dichte N_D) bleiben als ortsfeste *Raumladungen* in der *Raumladungszone* zurück (c). Die genauen Verhältnisse lassen sich durch die Integration der Poisson-Gleichung

$$\Delta\varphi = -\frac{\rho}{\varepsilon\varepsilon_0} \quad \text{bzw. eindimensional} \quad \frac{d^2\varphi}{dx^2} = -\frac{\rho(x)}{\varepsilon\varepsilon_0} \qquad (1.5)$$

berechnen. Einmalige Integration liefert den Feldverlauf (Felder außerhalb der Raumladungszone = 0), nochmalige Integration den Potentialverlauf (Potentialdifferenz = ursprüngliche Differenz der beiden Fermi-Energien).

Einstrahlung von Licht: Werden Lichtquanten eingestrahlt, deren Energie ausreicht, um Elektronen aus dem Valenzband ins Leitungsband anzuheben ($h\nu > E_\mathrm{L} - E_\mathrm{V}$), so werden zusätzliche Elektron-Loch-Paare gebildet. Geschieht dies im Bereich der Raumladungszone, werden sie durch das dort vorhandene Feld rasch getrennt (d), *Drift*-Strom fließt. Außerhalb der Raumladungszone ist die Trennung relativ unwahrscheinlich, da dort kein Feld vorhanden ist (e); ein geringer *Diffusions*-Strom kann fließen durch Ladungsträger, die zum p-n-Übergang diffundieren, überwiegend findet jedoch *Rekombination* statt. Durch intensivere Lichteinstrahlung werden die Potentialverhältnisse am p-n-Übergang merklich geändert, die Ladungsträgerkonzentrationen, die im Gleichgewichtsfall mit einer einheitliche Fermi-Energie berechnet werden konnte, werden nun formal durch zwei *Quasiferminiveaus* beschrieben (f). Die von der Photodiode gelieferte Spannung entspricht der Differenz der beiden Fermi-Energien.

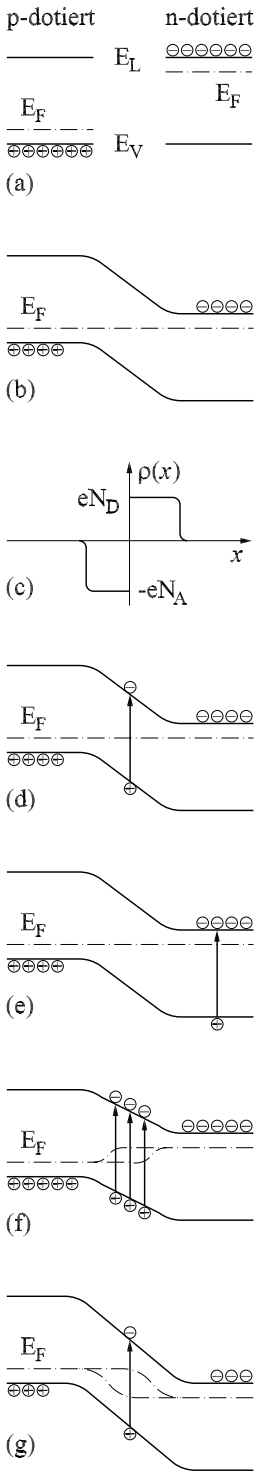


Abbildung 14: p-n-Photodiode, Bandstrukturen.

Die in Teilbild (f) dargestellten Potentialverhältnisse stellen sich bei *photovoltaischer* Verwendung einer Photodiode ein (Photoelement, Solarzelle). Bei Detektoranwendungen betreibt man die Diode meist mit angelegter Sperrspannung (g). Durch diese Betriebsart wird die Raumladungszone verbreitert und das dort vorhandene elektrische Feld erhöht. Beides verbessert die Detektoreigenschaften (größeres empfindliches Volumen, geringere Kapazität, schnellere Ladungsträgersammlung).

Kennlinie und Arbeitspunkt: Die Strom-Spannungs-Kennlinien einer Photodiode sind in Abbildung 15 skizziert. Bei Beleuchtung verschiebt sich die Kennlinie über einen sehr großen Bereich linear mit der Lichtintensität zu negativen Strömen.

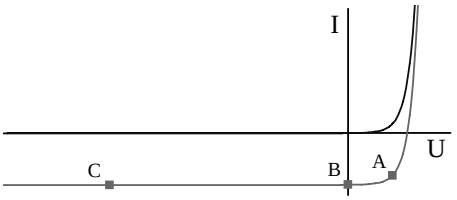


Abbildung 15: Strom-Spannungs-Kennlinien einer Photodiode ohne (obere Kurve) und mit Lichteinstrahlung (untere Kurve). A, B und C sind typische Arbeitspunkte.

Wichtige Arbeitspunkte sind:
A: Leistungsoptimierung, Diode als Spannungsquelle, Produkt aus Strom und Spannung möglichst groß, wird verwendet bei Solarenergieanwendungen.
B: Kurzschluss, einfachste Detektorbetriebsart, Strommessung ohne zusätzlichen Aufwand, der Kurzschlustrom ist proportional zur Lichtintensität.
C: Sperrspannung, die Diode wird in Sperr-Richtung betrieben, der Strom setzt sich aus Kurzschlussstrom (proportional zur Lichtintensität) und Sperrstrom zusammen. Standardbetriebsart insbesondere für p-i-n-Strukturen (Teilchendetektoren, vgl. 1.4.2).

Lawinenphotodioden: Bei hohen Sperrspannungen werden durch das große Feld in der Raumladungszone die Ladungsträger so sehr beschleunigt, dass bei Stößen weitere Ladungsträger angeregt werden, es kommt zum *Lawinendurchbruch*, der Sperrstrom erhöht sich drastisch. Die Diodenkennlinie knickt im Sperrbereich nach unten, d. h. zu hohen Sperrströmen hin ab. Dieser üblicherweise unerwünschte Effekt wird bei speziell dafür konstruierten Photodioden zur Stromverstärkung genutzt, die Verstärkung liegt zwischen 10 und 1000. Durch die zusätzliche Verstärkung in der Raumladungszone erreicht man mit Lawinenphotodioden in günstigen Fällen Einzelphotonenempfindlichkeit, dies mit deutlich höheren Quantenausbeuten (60...80 %) als bei Photomultipliern. Bei der Herstellung sind über die gesamte Detektorfläche sehr enge Toleranzen einzuhalten, damit der Verstärkungseffekt nicht zu sehr örtlich variiert, großflächige Detektoren sind dadurch nicht herstellbar. Eine interessante Anwendung (bei dem die Detektorfläche nur eine untergeordnete Rolle spielt) sind hochempfindliche Empfängerdioden bei der optischen Nachrichtenübertragung.

Halbleitermaterialien: Der für optische Anwendungen wesentlichste Materialparameter bei Halbleitern ist die Breite der verbotenen Zone, die Bandlücke (Bandgap, E_g). Sie bestimmt einerseits die langwellige Grenze des nutzbaren Wellenlängenbereichs, andererseits aber auch die Aktivierungsenergie für die thermische Anregung von Ladungsträgern und

damit den Dunkelstrom. Für eine große Empfindlichkeitsbandbreite muss man auch bei Sperrschichtphotodetektoren mit hohem Dunkelstrom, damit verbundenem hohen Rauschsignal bezahlen. Durch Kühlung kann man die Verhältnisse insbesondere bei langwelligen Detektoren deutlich verbessern. Einige gebräuchliche Materialien mit ihren Anwendungsbereichen sind in Abbildung 16 zusammengestellt.

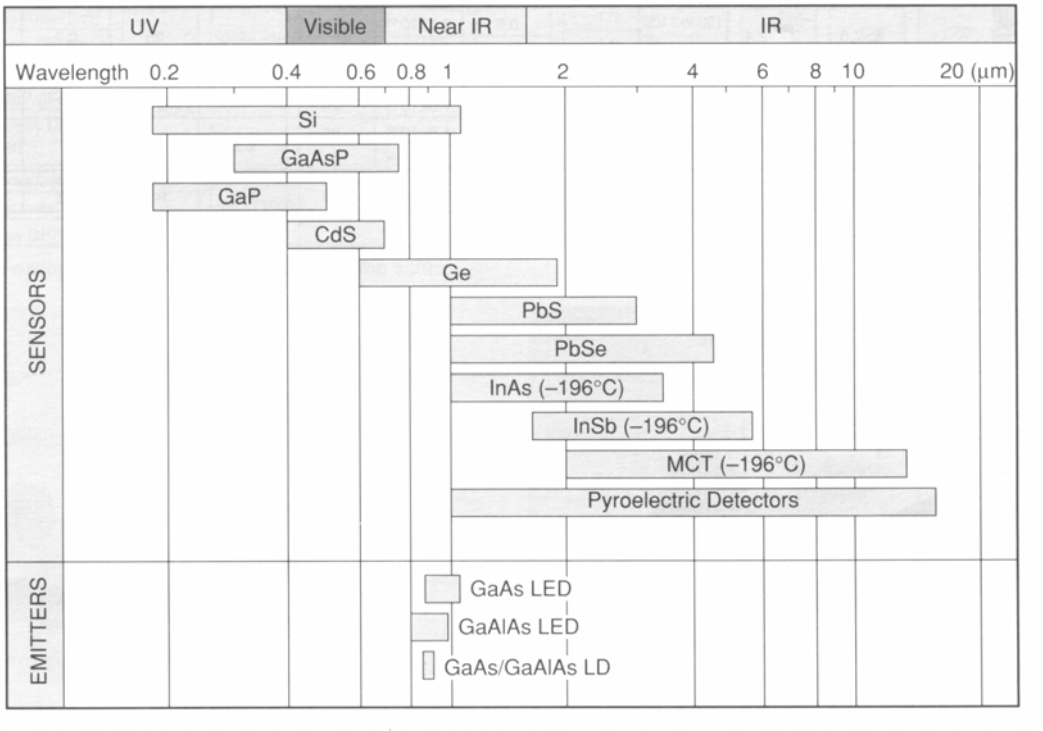


Abbildung 16: Anwendungsbereiche für verschiedene zur Herstellung von Photodetektoren verwendete Halbleitermaterialien; aus [10]. Als derzeit im Bereich der optischen Nachrichtenübertragung (1.3 oder 1.55 μm) sehr prominenter Halbleiter fehlt die III-V-Mischverbindung InGaAs mit einem Anwendungsbereich, der etwa dem von Ge entspricht. Die Abkürzung MCT steht für Mercury Cadmium Telluride *(*$Hg_{1-x}Cd_xTe$*), einer II-VI-Mischverbindung mit stark zusammensetzungsabhängiger Bandlücke.*

Der Verlauf des Absorptionkoeffizienten für Licht in der Nähe der Bandlücke ($h\nu \gtrapprox E_g$), d. h. nahe der langwelligen Empfindlichkeitsgrenze eines Halbleiters wird wesentlich von der Form der Energiebänder im k-Raum bestimmt. Bei Halbleitern mit *indirekter* Bandstruktur (Si, Ge) steigt der Absorptionskoeffizient zunächst nur mäßig an, bei solchen mit *direkter* (praktisch alle anderen) dagegen sehr steil. Für die Herstellung bedeutet das, dass bei Halbleitern mit indirekter Bandstruktur eine Optimierung für den jeweiligen Verwendungszweck erforderlich ist: gute Empfindlichkeit in der Nähe von E_g erfordert z. B. eine breite Raumladungszone. Einen Überblick über unterschiedlich optimierte Si-Photodioden eines Herstellers gibt Abbildung 17.

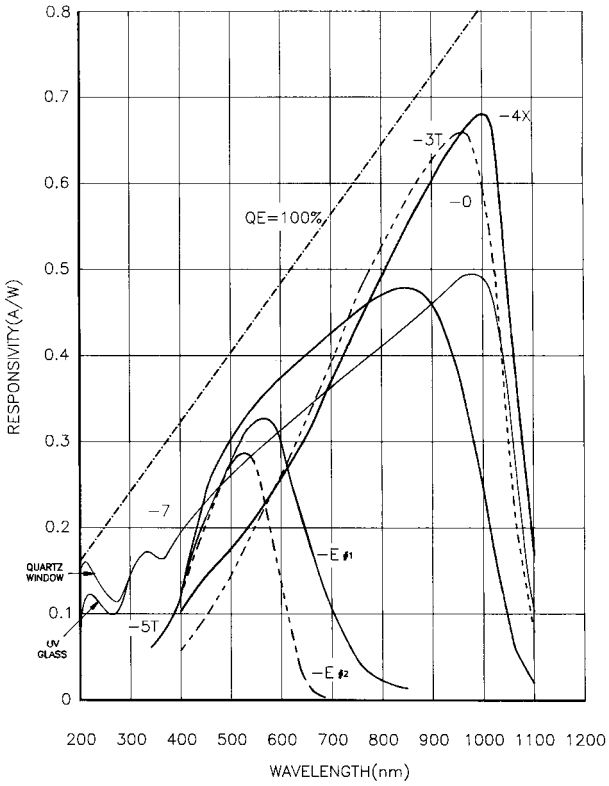


Abbildung 17: Empfindlichkeitskurven für unterschiedlich optimierte Silizium-Photodioden; aus [11]. Mit eingezeichnet ist die 'Idealkurve' für 100 % Quantenausbeute. Im UV-Bereich ist die Empfindlichkeit zusätzlich vom Fenstermaterial abhängig.

Bei Halbleitern mit direkter Bandstruktur ist eine solche Optimierung in der Regel nicht nötig, der Absorptionskoeffizient ist knapp oberhalb von E_g schon sehr hoch. Eine exemplarische Empfindlichkeitskurve für eine Photodiode aus einem Halbleiter mit direkter Bandstruktur (InGaAs) zeigt Abbildung 18.

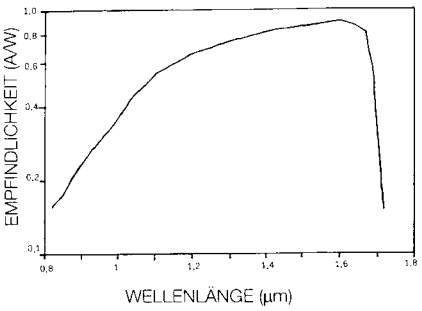


Abbildung 18: Typische Empfindlichkeitskurve für InGaAs-Photodioden (Halbleiter mit direkter Bandstruktur); aus [12].

Phototransistoren: Die Funktionsweise von Phototransistoren wird aus dem Ersatzschaltbild (Abbildung 19) deutlich: Der Photostrom einer Photodiode (beleuchtete Basis-Kollektor-Sperrschicht des Transistors) wird um den Faktor der Stromverstärkung des Transistors verstärkt. Der Signalstrom ist zwar wesentlich höher, die Linearität jedoch schlechter als bei Photodioden, da die Stromverstärkung vom Basisstrom abhängig ist.

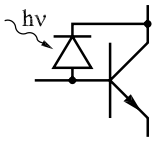


Abbildung 19: Phototransistor, Ersatzschaltbild.

Benötigt man eine Verstärkung des Messsignals mit guter Linearität direkt am Detektor, sollte man statt Phototransistoren besser integrierte Bausteine verwenden, bei denen eine Photodiode mit einem geeigneten Operationsverstärker im gleichen Gehäuse kombiniert ist.

Detektor-Arrays: Zur ortsaufgelösten Messung von Lichtintensitäten werden zunehmend ein- oder zweidimensionale Anordnungen von Photosensoren benutzt. Eindimensionale beispielsweise zur Messung eines kompletten Spektrums an einem Spektrometer, zweidimensionale vorzugsweise zur Bilderfassung. Als Detektormaterial wird wegen der dafür vorhandenen Technologie in erster Linie Silizium verwendet, zumindest für den sichtbaren Spektralbereich, allerdings nicht in bipolarer, sondern in MOS-Technologie. Für infrarotempfindliche Arrays sind neben Spezialmaterialien wie PtSi auch miniaturisierte Photoleiteranordnungen oder auch Mikrothermoelemente gebräuchlich. Zum Auslesen der Arrays werden zwei Techniken verwendet:

- Bei CCDs (*C*harge *C*oupled *D*evices) werden durch getakteten Ladungstransfer die in den Pixelelementen gesammelten Ladungen durch die Spalten und Reihen des Arrays zum Ausgang verschoben. Relativ billig in der Herstellung, nicht allzu genau.
- CMOS-Auslesearrays lesen dagegen jedes Pixelelement individuell durch eine Anordnung von einzelnen integrierten CMOS-Transistoren aus. Relativ aufwendig und teuer, aber sehr genau.

1.3.4 Photoleiter

Die Bedeutung von Photoleitern hat mit der Entwicklung neuer Materialien für Photodioden deutlich abgenommen. Eine neue Anwendung fanden sie jedoch in jüngster Zeit – miniaturisiert und integriert – im Bereich der infrarotempfindlichen Bildsensoren.

Photoleiter sind Materialien – in der Regel Halbleiter, deren Leitfähigkeit durch die optische Anregung von Ladungsträgern verändert wird. Diese Anregung kann über drei Prozesse erfolgen:

Intrinsische Photoleiter: Die Ladungsträger werden durch Anregung vom Valenzband ins Leitungsband erzeugt, dies entspricht dem Anregungsprozess bei Photodioden. Gebräuchliche Materialien sind III-V-Verbindungen wie InAs oder InSb und II-VI-Verbindungen wie CdS, $Hg_{1-x}Cd_xTe$ sowie PbS oder PbSe.

Extrinsische Photoleiter: Die Ladungsträgerdichte wird durch optische Übergänge von Störstellenzuständen ins Valenz- oder Leitungsband geändert. Ein klassisches Material war Germanium mit unterschiedlichen Dotierungen (Hg, Au, Cu, Ga); die minimale Anregungsenergie (Abstand vom Störstellenniveau zum Leitungs- bzw. Valenzband) – damit die langwellige Grenze des Detektors – ist abhängig vom Störstellentyp. Derzeit noch interessant ist dotiertes Silizium, da es mit moderner Technologie (*Planartechnik*) strukturierbar ist.

Intraband-Photoleiter: Durch optische Anregung werden Ladungsträger innerhalb eines Bandes zwischen Zuständen unterschiedlicher Beweglichkeit verschoben; es wird nicht die *Dichte* der Ladungsträger verändert, sondern deren *Beweglichkeit*. Die Leitfähigkeit ändert sich als Produkt aus Dichte und Beweglichkeit. Wenig verwendet, praktisch einziges bekanntes Material ist n-Typ-InSb.

1.3.5 Thermische Detektoren

Wie bei der Temperaturmessung können auch zur Messung von Strahlungsintensitäten (insbesondere im infraroten Wellenlängenbereich – *Wärme*-Strahlung) alle temperaturabhängigen physikalischen Größen (und das sind praktisch alle) genutzt werden. Der große Vorteil thermischer Detektoren liegt in dem weitgehend wellenlängenunabhängig konstanten Empfindlichkeitsverlauf, ein Nachteil in der gegenüber Quantendetektoren geringeren Maximalempfindlichkeit (vgl. Abbildung 6). Auch hier wieder nur eine Auswahl typischer Detektorprinzipien, Ausführlicheres liefert z. B. [6] und die darin zitierte Primärliteratur.

Pneumatische Detektoren: Ausgenutzt wird die thermische Ausdehnung von Gasen. Am bekanntesten ist die Golay-Zelle [13], die mit Xe gefüllt ist (Gas mit geringer Wärmeleitfähigkeit). Das Gasvolumen ist mit einer Membran abgeschlossen, deren Auslenkung optisch gemessen wird. Obwohl vor über 50 Jahren entwickelt, ist die Golay-Zelle immer noch einer der empfindlichsten IR-Detektoren für Raumtemperaturbetrieb.

Pyroelektrische Detektoren beruhen auf der Änderung der spontanen Polarisierung mit der Temperatur. Der Effekt tritt grundsätzlich bei allen Ferroelektrika auf. Man benutzt kondensatorähnliche Anordnungen, ein ferroelektrisches Material zwischen 2 aufgedampften Metallelektroden. Temperaturänderungen ändern die Polarisierung, mithin die zwischen den Elektroden messbare Spannung. Gebräuchliche Materialien sind Triglyzinsulfat (TGS), Strontium-Barium-Niobat (SBN), Lithiumniobat und Lithiumtantalat.

Bolometer sind temperaturabhängige Widerstände, zur Strahlungsmessung sind sie umso besser geeignet, je ausgeprägter die Temperaturabhängigkeit ihres Widerstandswerts ist. Verschiedene physikalische Konzepte sind naheliegend:

Speziallegierungen und *Metalloxide* in Dünnschichttechnik, dadurch mit kurzen Ansprechzeiten und geringer Wärmekapazität. Als Arrays teilweise auch in IR-Bildsensoren.

Supraleiter knapp unterhalb des Sprungpunkts, sehr empfindlich aber auch sehr aufwendig im Betrieb, da die Temperatur sehr genau konstant gehalten werden muss.

Kryo-Bolometer meist aus Halbleitern, die bei tiefen Temperaturen (4 K) sehr große Widerstandsänderungen zeigen.

Thermoelemente zeichnen sich durch Robustheit und einfache Handhabung aus, leider aber auch durch relativ geringe Empfindlichkeit. Durch Serienschaltung kann man die Größe des Messsignals deutlich erhöhen (Thermosäulen, meist in Dünnfilmtechnik), neuere Entwicklungen arbeiten mit miniaturisierten Anordnungen auf Halbleiterbasis (deutlich größere Thermokraft als bei Metallen).

1.4 Teilchen

Teilchendetektoren werden in der Hochenergiephysik, der Kernphysik und der Oberflächenphysik, außerdem als wichtige Nachweisgeräte in der Röntgentechnik verwendet. Unter dem Begriff 'Teilchen' sind einerseits Elementarteilchen und Ionen, andererseits aber auch Quanten elektromagnetischer Strahlung oberhalb einer nicht exakt definierbaren Mindestenergie zu verstehen. Die Funktionsprinzipien einiger typischer Detektortypen aus den Bereichen der Kern- und Oberflächenphysik werden kurz erläutert.

1.4.1 Szintillationszähler

Zu den ältesten Nachweistechniken für radioaktive Strahlung gehören *Szintillations*-Vorgänge, d. h. die Erzeugung von schwachen Lichtblitzen in geeigneten Materialien (Zinksulfid-Schirme zum Nachweis von Alpha-Strahlen etc.). Während in den Anfangszeiten der Kernphysik das dunkeladaptierte Auge des Experimentators eine große Rolle spielte, werden die Szintillationen inzwischen fast ausschließlich mit Photomultipliern nachgewiesen. Szintillatormaterial und -geometrie wählt man passend zur nachzuweisenden Strahlung, das Kathodenmaterial des Photomultipliers passend zum Emissionsspektrum des Szintillators.

Geladene Teilchen (Alpha-, Betastrahlung) regen im Szintillatormaterial – einem Isolator – durch Coulomb-Wechselwirkung Elektronen an und verlieren dadurch Energie. Gamma- oder Röntgenquanten verlieren einen Großteil oder ihre gesamte Energie in einem primären Stoßprozess (Compton- oder Photo-Effekt), das dabei erzeugte schnelle Elektron verhält sich wie ein Beta-Teilchen. Die zunächst hoch ins Leitungsband angeregten sekundären Elektronen thermalisieren zur Bandkante (Phononenanregung) und rekombinieren mit Defektelektronen an der Valenzbandoberkante. Mit gewisser Wahrscheinlichkeit wird dabei jeweils ein Lichtquant erzeugt; die Übergangswahrscheinlichkeit für diese *strahlende* Rekombination wird häufig durch Dotierungssubstanzen[4] erhöht, Thermalisierung und Rekombination laufen dann über geeignete Energieniveaus der Störstellen. Der mittlere Energieverlust pro generiertem Elektron-Loch-Paar ist über einen weiten Energiebereich eine nur vom Szintillatormaterial abhängige nahezu konstante Größe (typischerweise $10 \ldots 100\,\mathrm{eV}$). Damit ist die Gesamtzahl der Lichtquanten proportional zur Anfangsenergie des Teilchens, somit auch die Anzahl der aus der Kathode des Photomultipliers ausgelösten Elektronen und die Größe des Ladungsimpulses an der Anode[5]: Energiespektroskopie ist möglich.

Energieverlust pro Elektron-Loch-Paar, konkurrierende nichtstrahlende Rekombinationsprozesse, Lichtsammelwirkungsgrad und Quantenausbeute der Photokathode haben insge-

[4] Natriumjodid, das wichtigste Szintillatormaterial für Gammadetektoren, wird mit etwa 1 % Thallium dotiert, um eine kurze Lumineszenzabklingzeit, damit auch eine gute Lumineszenzausbeute zu erreichen.

[5] Durch eine geeignete Bauform ist sicherzustellen, dass das Szintillatorvolumen groß genug ist, damit das nachzuweisende Teilchen seine Energie vollständig verliert, und dass reproduzierbar alle Lichtquanten oder zumindest ein konstanter Bruchteil die Photokathode erreichen.

samt einen Energieaufwand von etwa 1 keV pro an der Photokathode erzeugtem Photoelektron zur Folge. Bei charakteristischen Teilchenenergien von 1 MeV entstehen somit ca. 1000 Photoelektronen. Deren Wahrscheinlichkeitsverteilung (Poisson- bzw. Gauß-Verteilung) begrenzt prinzipiell die Enegieauflösung der Szintillationszähler auf günstigstenfalls etwa 5 %, einen Wert, der deutlich schlechter ist als bei Halbleiterdetektoren. Für spektroskopische Anwendungen (Gammaspektroskopie) haben Szintillationszähler daher inzwischen nur noch eine geringe Bedeutung, wohl aber für solche Messanwendungen, bei denen es auf einfache Handhabung (keine Kühlung notwendig) oder große Detektorvolumina (Flüssig- oder Kunststoffszintillatoren) ankommt.

1.4.2 Halbleiterdetektoren

Halbleiterdetektoren zum Teilchennachweis werden als p-i-n-Diodenstrukturen gebaut. Die p- und n-Gebiete begrenzen eine undotierte eigenleitende (*i*ntrinsic) Zone (Abbildung 20). Durch eine an die Diode angelegte Sperrspannung wird ein elektrisches Feld erzeugt, das die vom ionisierenden Teilchen generierten Elektron-Loch-Paare quantitativ trennt, der Ladungsimpuls ist proportional zur Teilchenenergie bzw. zum Energieverlust. Da der mittlere Energieaufwand pro erzeugtem Elektron-Loch-Paar bei etwa 3 eV liegt, ist die Energieauflösung um mehr als eine Größenordnung besser als bei Szintillationszählern. Dicke von i-Schicht und Bedeckung sowie verwendetes Halbleitermaterial hängen von der Anwendung ab. Die Dicke der i-Schicht sollte der Reichweite der Teilchen entsprechen, die Dicke der Deckschicht deutlich kleiner sein. Für den Nachweis von Alpha- und Beta-Teilchen, teilweise auch für Röntgenstrahlung werden großflächige Silizium-Dioden verwendet, bei denen die i-Schicht nur wenige Mikrometer unter der Oberfläche liegt (Oberflächensperrschichtzähler). Für Gamma-Detektoren ist man dagegen auf Germanium angewiesen, das wegen seiner größeren Elektronendichte eine entsprechend größere Absorptionskonstante für die notwendigen Primärprozesse (Compton- und Photo-Effekt) aufweist. Bei diesen Detektoren liegt die Dicke der i-Schicht in der Größenordnung von Zentimetern. Der große Nachteil von Germanium ist seine große Leitfähigkeit bei Raumtemperatur, die Detektoren werden deshalb im Betrieb mit flüssigem Stickstoff (77 K) gekühlt.

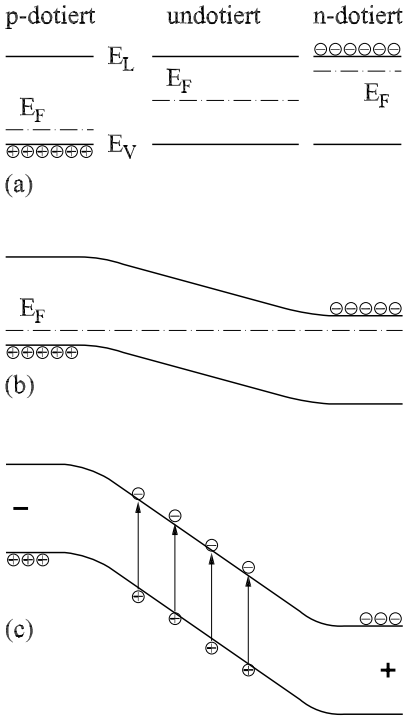


Abbildung 20: p-i-n-Detektor, Energiebandstrukturen:
E_L, E_V: Leitungs- und Valenzbandenergien.
(a) Lage der Fermi-Energien E_F in p-, i- und n-Halbleitern.
(b) Spannungslose p-i-n-Diode.
(c) p-i-n-Diode mit angelegter Sperrspannung; die in der i-Schicht generierten Elektron-Loch-Paare werden durch das starke Feld getrennt.

1.4.3 Sekundärelektronenvervielfacher

Zum hochempfindlichen Nachweis langsamer geladener Teilchen (Elektronen bei der Photoelektronenspektroskopie, Ionen in Massenspektrometern) werden offene Sekundärelektronenvervielfacher im Ultrahochvakuum eingesetzt. Sie entsprechen in Technik und Aufbau in etwa den in Photomultipliern verwendeten (vgl. 1.3.2); da jedoch keine Photokathode vorhanden ist, entfällt die Problematik der thermisch generierten Hintergrundereignisse, ein praktisch untergrundfreier Nachweis von Einzelereignissen ist möglich. Auch in diesen Anwendungsbereichen sind Mikrokanalplatten eine interessante Alternative zur herkömmlichen Technik, insbesondere dann, wenn deren Ortsauflösung zusätzliche physikalische Informationen liefern kann (Elektronenbeugung an Oberflächen).

2 Aktoren

Interessant an Messgrößen im physikalischen Experiment ist fast immer nicht nur ihr punktueller Wert, sondern auch ihre Abhängigkeit von variablen Messparametern. Solche gemessenen funktionalen Abhängigkeiten können dann auf Modellvorstellungen übertragen werden, die dadurch initialisiert, überprüft und weiterentwickelt werden. So werden im Bereich der Festkörperphysik aus der wellenlängenabhängig gemessenen Absorption oder Lumineszenz von Kristallen Aussagen über Energiezustände von Störstellen gemacht, aus der winkelabhängig gemessenen Einkopplung von Licht in Wellenleiter die optische Moden und der Brechungsindexverlauf berechnet, aus der temperaturabhängig gemessenen Dielektrizitätskonstanten oder der temperaturabhängig gemessenen optischen Frequenzverdopplung Informationen über strukturelle Phasenübergänge gewonnen, aus der winkel- und frequenzabhängig gemessenen Elektronenspinresonanz die Art und Symmetrie von Störstellen und deren Umgebung bestimmt.

Wünschenswert ist es immer, die Parameteränderungen weitgehend automatisiert – d. h. vom Rechner gesteuert – ablaufen zu lassen, um einerseits den Experimentator von ermüdender und fehleranfälliger 'Handarbeit' zu entlasten, andererseits den Verlauf des Experiments reproduzierbar, aber gleichzeitig einfach änderbar vorzugeben.

> Durch Rechnersteuerung soll vorrangig die Reproduzierbarkeit
> und damit die *Qualität* des Experiments verbessert werden.

2.1 Externe Geräte

Meist werden für die Einstellung von Parametern kommerzielle externe Geräte verwendet – Temperaturregler, optische Spektrometer, Netzgeräte für Magnetspulen, Hochspannungsversorgungen und viele andere. Üblicherweise sind diese Geräte mit *Standardschnittstellen* (RS 232, HPIB, USB) ausgestattet, über die – mehr oder weniger aufwendig – eine Steuerung möglich ist. Der Befehlsumfang ist fast immer sehr individuell auf das Gerät zugeschnitten, zur Erstellung eines Steuerungsablaufs ist ein genaues Studium der Gerätebeschreibung unerlässlich. In den ersten Generationen intelligenter Geräte wurden aus Performance-Gründen oft kryptische Binärkommandos verwendet, modernere Geräte dagegen 'hören' in der Regel auf Textkommandos, die ohne umständliche Kodierung erstellt werden können und weitgehend selbstdokumentierend sind.

Manche Aktoren lassen sich unter Umständen besser und schneller, immer jedoch billiger, relativ direkt vom Rechner ansteuern, nachstehend einige Beispiele.

2.2 Leistungsschalter

Um die Netzspannung von Elektrogeräten mittlerer und höherer Leistung potentialgetrennt zu schalten, werden Relais verwendet. Klassische elektromechanische Relais haben jedoch für die direkte Ansteuerung durch Rechner verschiedene Nachteile: relativ hohen Stromverbrauch, induktive Belastung, elektrische Störungen beim Schalten. Besser zu verwenden sind *Halbleiterrelais*, Bausteine mit dem in Abbildung 21 skizzierten Funktionsschema: mit einem geringen Steuerstrom (ca. 10 mA) werden zwei Leuchtdioden betrieben, die zwei entgegengesetzt gepolte Fototransistoren schalten[6]. Die reale Schaltung ist natürlich komplizierter: um Störungen zu verringern, wird üblicherweise dafür gesorgt, dass nur im Nulldurchgang der Wechselspannung ein- und ausgeschaltet wird, von der Nulldurchgangslogik werden als Lastschalter Thyristoren angesteuert.

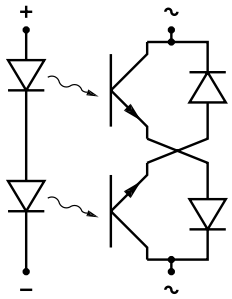


Abbildung 21: Halbleiterrelais, Funktionsschema.

Aufgrund des geringen Ansteuerstroms und eines weiten Bereichs (3. . . 30 V) der Ansteuerspannung können Halbleiterrelais problemlos an Standardschnittstellen des Rechners betrieben werden (Quittungsleitung der seriellen Schnittstelle, Einzelbit der Druckerschnittstelle). Mit geringem Aufwand können so auch Verbraucher im Kilowattbereich (Heizwiderstände in Kristallzuchtöfen o. ä.) vom Rechner geschaltet werden (Zweipunktregelungen, Puls-Pausen-Steuerungen).

2.3 Schrittmotoren

Mechanische Stellelemente im physikalischen Experiment (Verschiebetische, Hubtische, Drehtische), die im Mikro- bis Zentimeterbereich arbeiten, lassen sich relativ einfach mit dafür passenden Minimotoren ausrüsten und somit ferngesteuert bedienen. Zwei Systeme sind gebräuchlich: mit Inkrementalgebern gekoppelte Gleichstrommotoren und Schrittmotoren. Für beide gibt es komfortable komplette kommerzielle Steuergeräte, gerade aber bei Schrittmotoren ist es naheliegend, die Steuerung direkt dem Rechner zu übertragen.

Beim Schrittmotor wird ein magnetisierter Anker von einem durch geeignete Spulenströme erzeugten Drehfeld weitergedreht. Im Gegensatz zum *Synchronmotor* dreht sich das Magnetfeld in diskreten Schritten und mit beliebiger, auch wechselnder, Geschwindigkeit (bis zu einer durch die Bauart bedingten Höchstgeschwindigkeit). Die Winkelauflösung ist primär durch die *Polzahl* festgelegt, Werte zwischen 10 und 1000 sind gebräuchlich, bei Vollschrittbetrieb bedeutet das eine entsprechende Anzahl von Schritten pro ganzer Umdrehung. Die typischen Betriebsarten von Schrittmotoren sind in den Abbildungen 22, 23 und 24 skizziert. Zur einfacheren Darstellung des Prinzips ist die Polzahl auf 4 reduziert.

[6] **Wichtiger Sicherheitsaspekt:** Wie die Prinzipschaltung zeigt, schalten Halbleiterrelais nur einpolig ab, außerdem ohne wirkliche mechanische Trennung. Auch im abgeschalteten Zustand fließt noch ein kleiner Versorgungsstrom für die Schaltung. Wenn an einem so abgeschalteten Gerät gearbeitet wird, muss eine weitergehende Abtrennung der Stromversorgung erfolgen, beispielsweise durch einen zusätzlichen mechanischen Schalter.

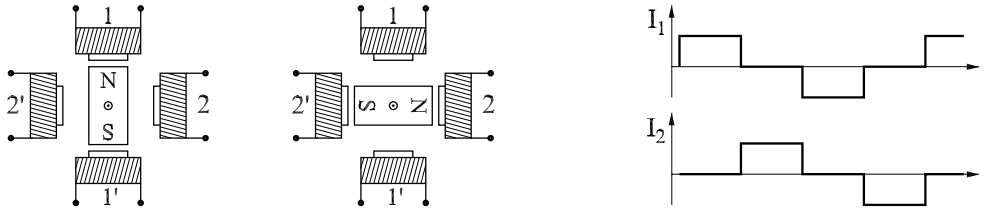


Abbildung 22: ***Vollschrittbetrieb:*** *Der Anker dreht sich bei einem Schritt zum nächsten Pol, im skizzierten Fall eines 4-Pol-Motors entspricht dies einer Rotation um 90°. Im rechten Teilbild der Verlauf der beiden Spulenströme für eine volle Umdrehung (4 Schritte), die Spulen 1 und 1' bzw. 2 und 2' sind jeweils geeignet in Serie geschaltet (⇒ Zwei-Phasen-Motor, bipolare Betriebsart).*

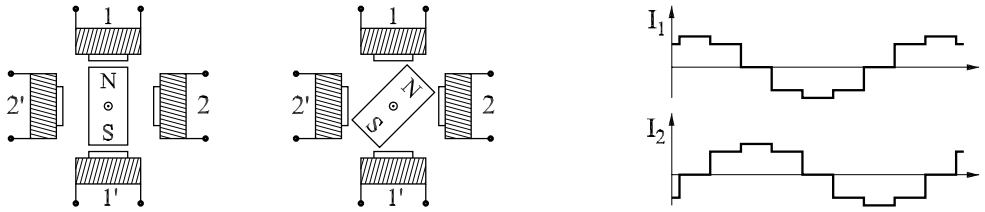


Abbildung 23: ***Halbschrittbetrieb:*** *Der Anker dreht sich bei einem Halbschritt um einen halben Polabstand weiter (hier 45°). Rechts der Stromverlauf (8 Halbschritte pro voller Umdrehung).*

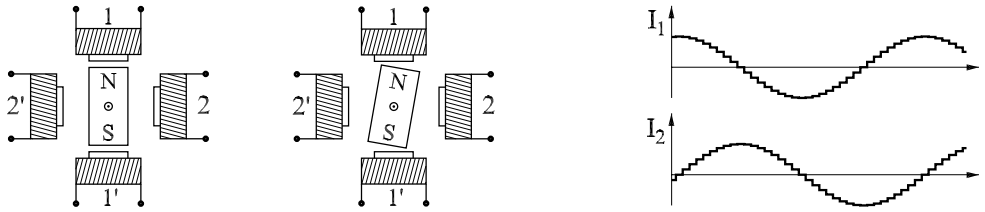


Abbildung 24: ***Mikroschrittbetrieb:*** *Der Anker dreht sich bei jedem Mikroschritt um einen Bruchteil des Polabstands weiter (hier 1/9 ≙ 10°). Die Stromverläufe nähern sich Sinus- bzw. Cosinusfunktionen an. Mikroschrittbetrieb setzt voraus, dass der Schrittmotor von seiner Bauform her dafür geeignet ist.*

In den Abbildungen 22, 23 und 24 wird angenommen, dass der Schrittmotor jeweils *bipolar* betrieben wird. Der Spulenstrom nimmt positives und negatives Vorzeichen an. Dadurch kommt man mit 2 Phasen aus, benötigt aber etwas aufwändigere Treiberendstufen als bei *unipolarer* Betriebsart. Diese ist in Abbildung 25 schematisiert (dort für Vollschrittbetrieb, Halbschrittbetrieb ist in ähnlicher Weise wie bei der bipolaren Betriebsart möglich, grundsätzlich auch Mikroschrittbetrieb). Bei bipolarer Betriebsart werden (mindestens) 3 Zustände der Treiberstufen benötigt (positiv, null, negativ), bei unipolarer Betriebsart nur 2 (Strom, stromlos). Abbildung 26 zeigt die schematisierte Schaltung der jeweiligen Treiberendstufen.

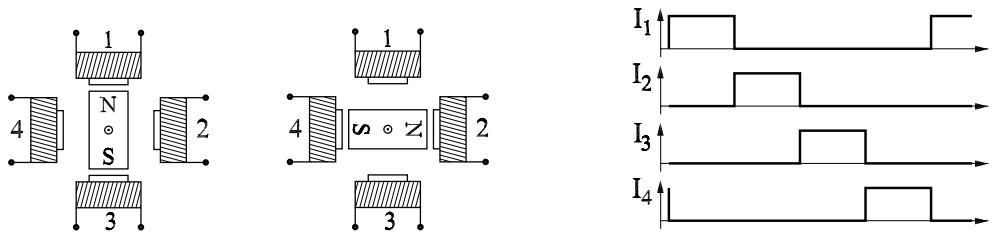


Abbildung 25: ***Unipolare Betriebsart:*** *Bei dieser Betriebsart führen die vier dargestellten Spulen jeweils einzeln Strom. Rechts der zeitliche Verlauf der 4 Spulenströme.*

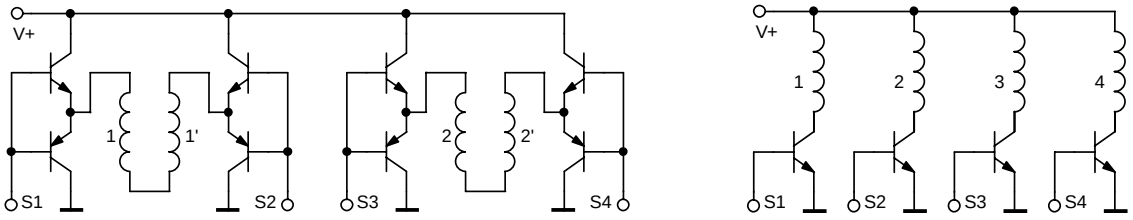


Abbildung 26: Treiberendstufen (schematisiert) für bipolaren (links) und unipolaren Betrieb (rechts) von Schrittmotoren. S1...S4 sind die Steuereingänge.

Rechnersteuerung: Für die Ansteuerung von Schrittmotoren durch den Rechner bieten sich vier Varianten an:

Intelligente Steuergeräte erhalten vom Rechner eine Zielvorgabe (n Schritte vorwärts) und erledigen das dazu notwendige selbständig.

Einfache Steuergeräte erwarten vom Rechner *Takt-* und *Richtungs*-Impulse (meist TTL-kompatibel) und generieren nur die zugehörige Spulenstromabfolge. Jeder einzelne Motorschritt muss vom Rechner veranlasst werden.

Treiberstufen mit Darlington-Transistoren (als ICs mit 8fach Treibern erhältlich) lassen sich über eine Parallel-Ein/Ausgabe-Karte oder über den Druckerausgang des Rechners ansteuern. In diesem Fall muss das Steuerprogramm die Abfolge der Ströme für die einzelnen Spulen (Abbildung 25) als Binärwerte erzeugen und sich den jeweiligen Status merken.

D/A-Wandler mit Stromverstärkern können verwendet werden, um einen rechnergesteuerten Mikroschrittbetrieb zu realisieren.

Geschwindigkeit: Wichtig ist es, die spezifizierte *Start-Stop*-Geschwindigkeit (bauartabhängig zwischen etwa 100 und 1000 Hz) nicht zu überschreiten, da nur dann eine schrittgenaue Positionierung gewährleistet ist. Wenn größere Wege zurückzulegen sind, kann – falls nötig – die Geschwindigkeit in einer definierten Beschleunigungsphase (mit einer dazu korrespondierenden Bremsphase) bei fast allen Motoren auf das fünf- bis zehnfache erhöht werden.

Stromabsenkung: Bei Voll- und Halbschrittbetrieb ist es sinnvoll, im Ruhezustand den Spulenstrom auf einen niedrigeren *Haltestrom* abzusenken, die meisten Steuerungen se-

hen einen solchen Betrieb vor (zusätzliche Steuerleitung). Dies führt zu einer deutlich geringeren thermischen Belastung des Motors und vor allem auch der Umgebung.

2.4 Servos

Vor allem für Anwendungen in der Fernsteuerung (Modellbau, mechanisches Spielzeug, Robotik) wurden kompakte Stellmotoren entwickelt, die mit einigermaßen standardisierten Signalen angesteuert werden können. Diese sogenannten *Servos* können auch in Experimenten überall dort eingesetzt werden, wo einfache Verstellaufgaben automatisiert werden sollen (Blenden, Shutter, Klappspiegel). Servos bestehen aus kleinen leistungsfähigen Motoren mit einem Untersetzungsgetriebe, über das eine Welle oder Scheibe am Ausgang gedreht wird. Deren Winkelstellung wird mit einem Drehwiderstand gemessen und dem Ansteuersignal entsprechend eingestellt (geregelt). Als Winkelverstellbereich ist etwa eine halbe Umdrehung üblich. Das Ansteuersignal besteht aus Impulsen mit einer festen Folgefrequenz (oft 50 Hz) und variabler Länge (z. B. 1…2 ms). Die Pulslänge legt die Winkelposition innerhalb des Verstellbereichs fest (Abbildung 27). Das Steuersignal braucht nur kurzzeitig angelegt zu werden, der Servo behält danach die vorgewählte Position bei.

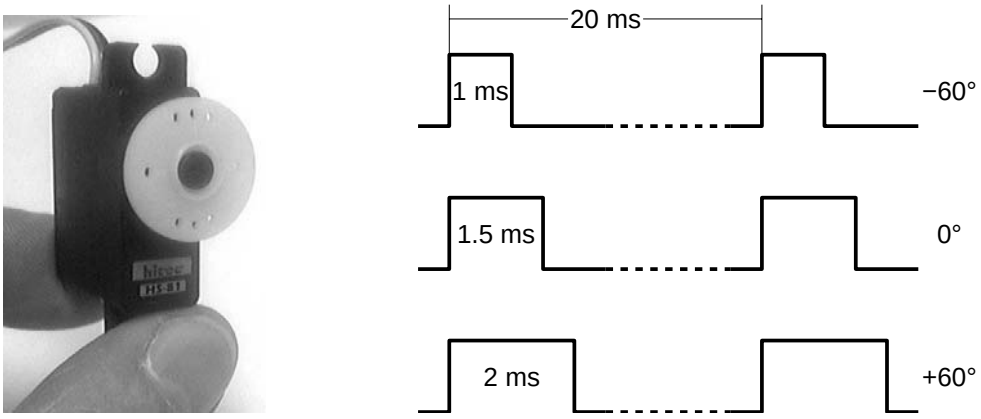


Abbildung 27: Links ein typischer Servo aus dem Fernsteuerungsbereich. Rechts die Ansteuersignale (TTL-kompatibel zwischen ≈0 V und > 3 V wechselnd) für die angegebenen Winkelpositionen.

2.5 Altgeräte-Recycling

Alte Matrixdrucker oder Flachbettplotter, die in vielen Labors noch vorhanden sind, können im Rahmen ihrer Möglichkeiten sehr gut als ein- bzw. zweidimensionale mechanische Stellelemente eingesetzt werden. Die Ansteuerung ist einfach, sie können über die Drucker- oder die serielle Schnittstelle des Rechners betrieben werden, Drucker mit nahezu standardisierten Escape-Sequenzen, Plotter mit HP-GL (*H*ewlett *P*ackard *G*raphics *L*anguage). Ein Drucker kann so zum Beispiel noch dazu verwendet werden, Filter in

optischen Strahlengängen zu wechseln, aus einem Plotter kann mit Hilfe von zwei Umlenkspiegeln ein optischer Scanner werden.

2.6 Piezostellelemente

Mechanische Bewegungen im Mikro- bis Nanometerbereich lassen sich mit Piezoaktoren ausführen. Ihre Funktion beruht auf dem piezoelektrischen Effekt (genauer dessen Umkehrung): Polare Festkörper reagieren auf ein elektrisches Feld mit Längenänderung oder Scherverformung. Typische Bauformen von dreidimensionalen Piezoverstellern, wie sie beispielsweise bei der Rastersondenmikroskopie eingesetzt werden, zeigt Abbildung 28.

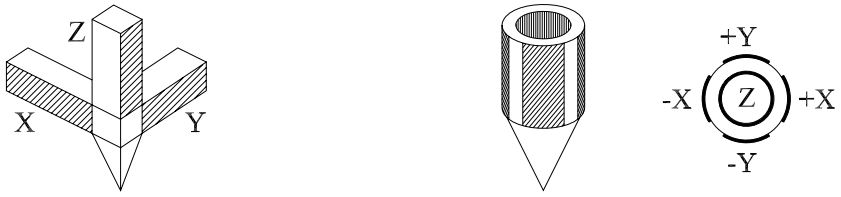


Abbildung 28: Bauformen von dreidimensionalen Piezoverstellern: Dreibein (links) und Biegeröhrchen (rechts, ganz rechts Schnitt durch die Elektrodenanordnung). Beim Biegeröhrchen bewirken symmetrische Spannungsänderungen an den X- oder Y-Elektroden eine Verbiegung, eine Spannungsänderung an der Z-Elektrode eine Längenänderung.

Mit einfachen keramischen Piezostellelementen können Bewegungen im Mikrometerbereich mit Auflösungen im Nanometerbereich erreicht werden. Die notwendigen Betriebsspannungen liegen bei einigen hundert Volt; eine direkte Ansteuerung vom Rechner ist über eine Kombination von D/A-Wandler und Spannungsverstärker möglich. Bei sehr genauen Anwendungen sollte die thermische Ausdehnung von Piezokeramik und Halterung durch eine geeignete Lageregelung kompensiert werden (Regelung auf konstanten Tunnelstrom bei der Rastertunnelmikroskopie, auf maximales Signal bei piezobetriebenen Fabry-Perot-Interferometern).

2.7 Entstörung

Beim Schalten größerer elektrischer Leistungen, aber auch beim Betrieb von Schrittmotoren[7] können elektromagnetische Störfelder auftreten, die sinnvolle Messungen im Experiment erschweren, manchmal verhindern. Solche Störungen sollten schon an der Störquelle soweit wie möglich reduziert werden. Die dazu nötigen Maßnahmen hängen von der Art der Störung ab, diese sollte zunächst analysiert werden. Eine unvollständige Auflistung möglicher Entstörmaßnahmen:

[7] Einfache Schrittmotorsteuergeräte stellen den Motorstrom durch eine Puls-Pausen-Taktung der Betriebsspannung ein. Da diese mit deutlich höherer Frequenz als der maximalen Schrittfrequenz erfolgen muss, sind relativ steile Schaltflanken die Folge.

Kurze elektrische Verbindungsleitungen: Trivial, aber immer wieder vergessen, je weniger (Sende- oder Empfangs-) Antenne, umso besser.

Erdung: Wichtig, kann aber auch stören, wenn Erdschleifen auftreten; oft hilft dann, alle Erdleitungen sternförmig zusammenzuführen.

Abschirmung: Abgeschirmte Kabel und Metallgehäuse gegen elektrische, weichmagnetische Materialien hoher Permeabilität (sehr aufwendig) gegen magnetische Felder.

Symmetrisierung: Ferritringe um Steuerkabel sorgen für eine Hochfrequenzsymmetrisierung und erniedrigen deutlich die HF-Abstrahlung ($\rightsquigarrow$ Standard bei hochwertigen Monitor- oder SCSI-Kabeln).

Potentialtrennung: Eine einfache Trennung durch Optokoppler vermeidet Probleme mit Erdschleifen und mit Potentialunterschieden zwischen Geräten.

Optische Verbindungen: Eine Signalübertragung via Lichtleiter ist zwar etwas aufwendig, aber an Störsicherheit kaum zu überbieten.

3 Signalverarbeitung

Detektoren und Sensoren wandeln physikalische Messgrößen in korrespondierende elektrische Größen wie Strom, Spannung, Ladung oder Widerstand um. Die Signalverarbeitungselektronik muss in ihrer Eingangsschaltung gezielt für diese spezifische elektrische Größe ausgelegt sein.

> Bei der Signalverarbeitung ist zunächst zu überlegen, in welcher Signaleigenschaft sich die gesuchte physikalische Messgröße etabliert.

Analoge elektrische Größen wandelt man zunächst in eine dazu proportionale Spannung, Ereignisgrößen in geeignete einheitliche Zählimpulse. Einige der dazu verwendeten Schaltungsprinzipien werden nachfolgend anhand von vereinfachten Operationsverstärkerschaltungen kurz skizziert (eine vertiefte Einführung in die Halbleiterschaltungstechnik mit detaillierter Diskussion der Eigenschaften typischer Operationsverstärker und ihrer Schaltungsdimensionierung findet sich z. B. in [14] oder [15]).

3.1 Strom

Viele Detektoren und Sensoren liefern ein Stromsignal, das der physikalischen Messgröße proportional ist (Anodenstrom bei Photomultiplieren, Sperrstrom bei Photodioden). Dieser Strom wird mit einem als Strom-Spannungs-Wandler beschalteten Operationsverstärker[8] (Abbildung 29) in ein Spannungssignal $U = -R \cdot I$ umgesetzt.

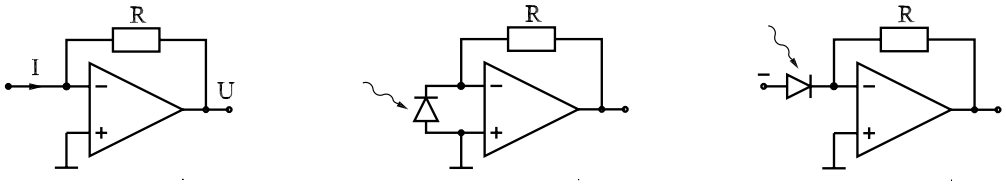


Abbildung 29: Strom-Spannungs-Wandler: Links die Grundschaltung, in der Mitte mit einer Photodiode im Kurzschlussbetrieb, rechts im Sperrbetrieb.

Bei Photodioden sind die beiden skizzierten Schaltungen gebräuchlich, welche man im Einzelfall wählt, hängt von den experimentellen Anforderungen ab:

Im *Kurzschlussbetrieb* fließt kein Dunkelstrom, interessant für den rauscharmen Nachweis sehr geringer Lichtintensitäten.

Im *Sperrbetrieb* ist die Diodenkapazität geringer, das Feld in der Raumladungszone größer als im Kurzschlussbetrieb, ideal für höhere Signalbandbreiten; allerdings fließt auch ohne Beleuchtung ein – wenn auch geringer – Dunkelstrom.

[8] Verschiedene Hersteller bieten Bausteine an, bei denen Photodiode und Operationsverstärker im gleichen Gehäuse integriert sind.

3.2 Spannung

Zur Spannungsverstärkung wird der Operationsverstärker *nichtinvertierend* betrieben (Abbildung 30). Diese Betriebsart hat den Vorteil eines sehr hohen Eingangswiderstands, belastet somit die Signalquelle nur minimal. Bei sehr hochohmigen Quellen und/oder kleinen Signalpegeln sollte man Operationsverstärker mit FET-Eingang verwenden, der Eingangswiderstand ist deutlich größer, das Eingangsrauschen deutlich kleiner als bei konventionellen mit bipolarem Eingang.

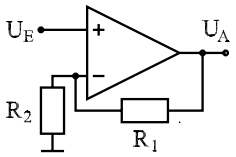


Abbildung 30: Spannungsverstärker.

Die Verstärkung V wird wie üblich aus den idealen Eigenschaften des Operationsverstärkers (Eingangsspannungdifferenz und Eingangsstrom gleich null) berechnet:

$$V = \frac{U_\mathrm{A}}{U_\mathrm{E}} = 1 + \frac{R_1}{R_2} \tag{3.1}$$

Ist nur eine Impedanzwandlung, keine Signalverstärkung notwendig, erreicht man dies durch $R_1 = 0$ (Spannungsfolger).

3.3 Widerstand

Die Widerstandsmessung wird mit Hilfe einer Konstantstromquelle auf eine Spannungsmessung zurückgeführt[9]. Um die Messung wenig zu stören, insbesondere auch um wenig Wärme zu produzieren, sollte der Betriebsstrom möglichst niedrig eingestellt werden. Trotzdem kann bei Widerstandsdetektoren eine langsame zeitliche Veränderung (Drift) im Widerstandswert zu Messungenauigkeiten führen. Eine Verbesserung erreicht man in solch einem Fall meist durch eine modulierte Messung: Ein Messparameter wird periodisch verändert (bei optischen Messungen zum Beispiel die Intensität des Anregungslichts), das zugehörige Wechselspannungssignal wird gemessen, die (langsame) Gleichspannungsdrift wird durch einen Hochpass – im einfachsten Fall ein Koppelkondensator – unterdrückt. Noch besser geht's mit dem im folgenden Abschnitt beschriebenen Lock-In-Verfahren, einer Korrelationsmesstechnik.

3.4 Lock-In-Verfahren

Die Lock-In-Technik ist eines der wichtigsten Korrelationsmessverfahren: Das zu messende Signal wird geeignet moduliert, die Korrelationsfunktion mit der Modulation wird ausgewertet. Im Prinzip wird eine sehr schmalbandige Messung bei der Modulationsfrequenz ausgeführt, dadurch werden insbesondere niederfrequente Anteile im Rauschen (Drift) effizient unterdrückt. Darüber hinaus kann für die Modulation ein Frequenzbereich gewählt werden, in dem das Rauschen minimal ist. Eine Signalmodulation kann auf unterschiedlichste Art realisiert werden, bei optischen Experimenten (Absorptions- oder Lumineszenzmessungen) wird das Anregungslicht moduliert, bei Spinresonanzexperimenten das

[9] Auch möglich, aber nicht üblich ist die Messung des reziproken Widerstands über den Strom (Konstantspannungsquelle).

Magnetfelds (allerdings nur geringfügig). Das so modulierte, meist stark verrauschte Signal wird *phasenrichtig* gleichgerichtet – wie in Abbildung 31 schematisch dargestellt durch abwechselnde Multiplikation mit +1 oder -1. Diese phasenrichtige Multiplikation wird durch eine mit der Modulation korrelierte Referenzspannung gesteuert. Der Integrator am Ausgang sorgt für die gewünschte Bandbreitenbegrenzung.

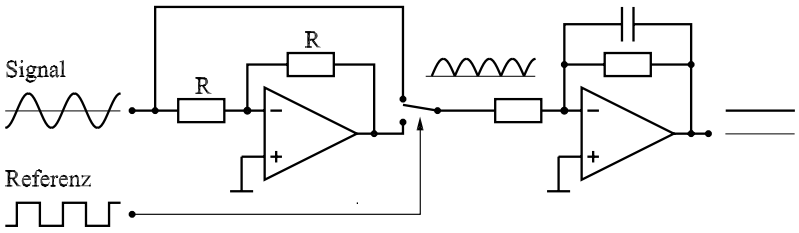


Abbildung 31: Lock-In-Verstärker, Funktionsprinzip.

Eine Simulation der Wirkungsweise (durchgeführt mit MATLAB®) ist in Abbildung 32 dargestellt. Das linke Teilbild zeigt das (ideale) Signal und die beiden bei der Simulation verwendeten Rauschspannungen, im mittleren Teilbild die Messergebnisse bei Anwendung eines einfachen Tiefpassfilters auf die verrauschten Signale, rechts die Messergebnisse bei zusätzlicher Anwendung des Lock-In-Verfahrens.

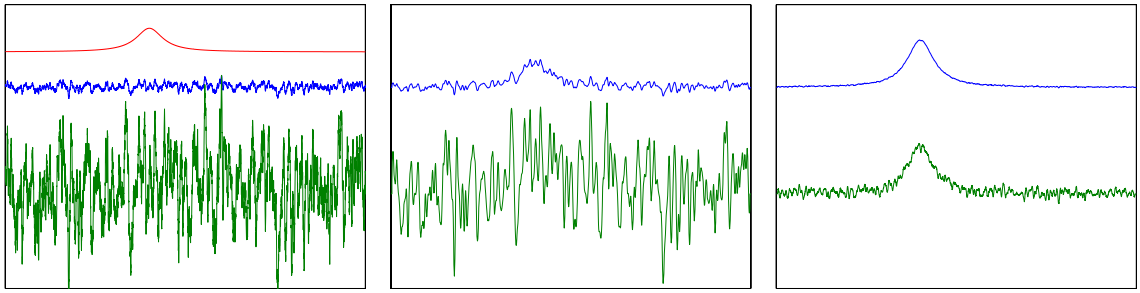


Abbildung 32: Lock-In-Verstärker, Simulation. Links: Ausgangssignale. Mitte: Tiefpassfilterung. Rechts: Lock-In-Verfahren.

3.5 Ladung

Bei Teilchendetektoren ist die elektrische Ladung pro nachgewiesenem Teilchen die primär interessierende Größe, zumindest dann, wenn der Detektor zur Energiespektroskopie eingesetzt wird (Szintillationszähler und Halbleiterdetektoren in der Alpha- oder Gamma-Spektroskopie). Die Ladung ist das Integral über einen Stromimpuls, eine ladungsproportionale Spannung erzielt man mit einem als Integrator beschalteten Operationsverstärker (Abbildung 33).

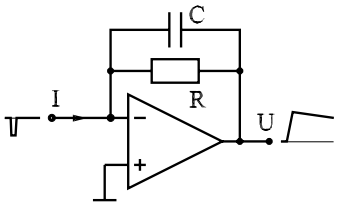


Abbildung 33: Integrator als Ladungs-Spannungs-Wandler.

Für die Ausgangsspannung des Integrators gilt die Differentialgleichung

$$RC \cdot \frac{dU}{dt} + U + R\,I = 0 \tag{3.2}$$

mit der allgemeinen Lösung

$$U = -R\,I + K \exp\left(-\frac{t}{RC}\right). \tag{3.3}$$

Im idealisierten Fall eines Rechteckimpulses der Länge τ, d. h. $I = 0$ für $t < 0$, $I = I_0$ für $0 < t < \tau$ und $I = 0$ für $t > \tau$ ergibt sich für die Spannung am Ende des Impulses

$$U_{\text{peak}} = -R\,I_0 + R\,I_0 \exp\left(-\frac{\tau}{RC}\right) \tag{3.4}$$

und mit der Reihenentwicklung der Exponentialfunktion

$$U_{\text{peak}} = -\frac{I_0\tau}{C}\left(1 - \frac{\tau}{RC} + \ldots - \ldots\right). \tag{3.5}$$

Gute Ladungsproportionalität (Ladung $= I_0\tau$) ist folglich dann gewährleistet, wenn die Schaltung so ausgelegt wird, dass für typische Signalimpulse $RC \gg \tau$ gilt. Allerdings sollte die Integrationszeitkonstante RC auch nicht größer gewählt werden als für die erforderliche Genauigkeit nötig, da ansonsten *pile-up*-Probleme bei höheren Impulsraten auftreten.

3.6 Ereignis

Sollen Einzelereignisse nur gezählt werden (photon counting u. ä.), ohne dass weitere Signalinformationen auszuwerten sind, werden – gegebenenfalls nach Verstärkung in einem schnellen Verstärker – die Signalimpulse in einem als Schwellwertdiskriminator fungierenden Komparator zu einheitlichen Zählimpulsen geformt (Abbildung 34).

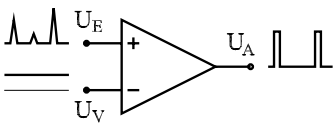


Abbildung 34: Komparator als Diskriminator.

Der Komparator hat neben der Impulsformung die Funktion, Rauschimpulse – insbesondere vom Verstärkerrauschen herrührend – zu unterdrücken (Impulse, die kleiner als U_{V} sind, werden zurückgehalten).

3.7 Zeit

Bei vielen Messgrößen ist die zeitliche Entwicklung – z. B. nach einer gepulsten Anregung – interessant, da dadurch die Dynamik von Prozessen untersucht werden kann. Geht es dabei um längere Zeiträume (Zerfallszeiten von Hologrammen), kann die Zeitmessung über ein Rechnerprogramm realisiert werden, bei kurzen Zeiten dagegen sind in der Regel spezialisierte Messgeräte erforderlich.

3.7.1 Transientenspeicher

Die Entwicklung schneller Analog-Digital-Wandler und entsprechend schneller Speicher ermöglicht die direkte Erfassung von zeitabhängigen Analogsignalen in Digitalspeicheroszilloskopen oder *Transientenrecordern*. Es können sowohl einmalige Vorgänge gespeichert wie auch – zur Rauschunterdrückung – sich wiederholende Messsignale aufsummiert werden. Die möglichen Zeitauflösungen erreichen den Subnanosekundenbereich.

Ohne Analog-Digital-Wandler, dafür mit extrem schneller *Add-One*-Arithmetik, gibt es die gleiche Technik zur zeitaufgelösten Ereigniszählung (*Multi-Channel Scaler*).

3.7.2 Boxcar-Technik

Eine relativ alte Messtechnik, die bei sich wiederholenden Signalen angewendet werden kann, ist die Boxcar-Technik. Das zeitabhängige Signal wird in einem kurzen Zeitfenster abgetastet, das langsam über das Signal verschoben wird (Abbildung 35). Ein Tiefpassfilter (Integrator) am Ausgang glättet das zunächst kammförmige Abtastsignal zu einer langsam veränderlichen zeitabhängigen Ausgangsspannung (Y). Ein zweiter Ausgang liefert eine Spannung, die der Zeitverschiebung der Abtastung proportional ist (X).

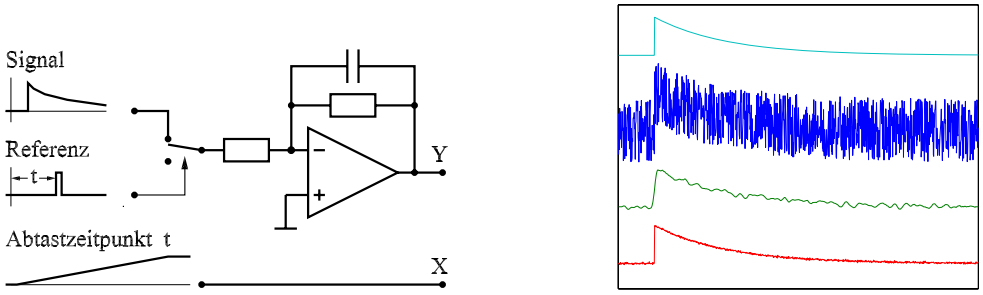


Abbildung 35: Boxcar-Verfahren, links Funktionsschema, rechts Simulation.

Die Simulation des Verfahrens (wieder durchgeführt mit MATLAB®) zeigt die Wirkung – rechtes Teilbild der Abbildung 35: Oben unverrauschtes und verrauschtes Idealsignal, darunter das verrauschte Signal nach Boxcar-Abtastung und -Integration. Ganz unten zum Vergleich dasselbe Signal mit einem summierenden Transientendigitalisierer integriert (bei gleicher Messzeit und hier in der Simulation sehr viel besserer Zeitauflösung ist das Rauschen deutlich geringer als bei der Boxcar-Technik).

Abgesehen von der Zeitmessung kann das Boxcar-Verfahren – dann mit festem Abtastzeitpunkt – als Korrelationsmesstechnik bei gepulst angeregten Experimenten (nichtlineare Optik mit Pulslasern) eingesetzt werden. Gemessen wird nur während der Pulsanregung, das Rauschen dazwischen wird unterdrückt. Digital, d. h. in Verbindung mit Ereigniszählung (photon counting), ist dies derzeit eines der empfindlichsten Messverfahren.

3.7.3 Zeit-Impulshöhen-Wandlung

Die zeitliche Verzögerung seltener Zählereignisse gegenüber einem Anregungsimpuls kann durch Zeit-Impulshöhen-Wandlung mit relativ guter Zeitauflösung bestimmt werden. Bei dieser Technik wird eine analoge Größe, die Zeitdifferenz zwischen zwei Impulsen, in eine dazu proportionale andere, einen Spannungswert, gewandelt, indem während der zu messenden Zeit eine Kapazität mit konstantem Strom aufgeladen wird. Der dabei erreichte Spannungswert ist ein Maß für die Zeitdifferenz. Eine Auftragung der Häufigkeitsverteilung der gemessenen Zeiten ($\rightsquigarrow$ A/D-gewandelte Spannungswerte) ergibt den Zeitverlauf des durch die Zählereignisse repräsentierten Signals. Da nach einem Anregungsimpuls das jeweils erste Stop-Ereignis zum Tragen kommt, eignet sich die Methode nur für seltene Ereignisse, da ansonsten kurze Zeiten überbewertet werden.

4 D/A- und A/D-Wandler

Experimentelle Steuergrößen (Heizstrom, elektrisches Feld, ...), die vom Rechner vorgegeben werden sollen, müssen dazu meist zuerst in Analogwerte umgesetzt werden. Diese Digital/Analog-Wandlung erfolgt in digital ansteuerbaren Peripheriegeräten oder auf PC-Karten durch spezielle Bausteine – D/A-Wandler. Experimentelle Messwerte andererseits (Spannung, Widerstand, Temperatur, Lichtintensität, Druck, ...) liegen am Experiment im allgemeinen analog vor und müssen – gegebenenfalls nach Umwandlung in elektrische Größen – zur Bearbeitung durch den Rechner in Digitalwerte umgeformt werden. Auch diese Analog/Digital-Wandlung erfolgt – in externen Messgeräten wie Digitalvoltmetern oder auf PC-Karten – durch spezielle Bausteine, A/D-Wandler.

> Die Grenze zwischen Digital und Analog ist auch eine Grenze zwischen scheinbarer Exaktheit und realer Ungenauigkeit.

Im folgenden sollen die Verfahren, nach denen die Wandlerbausteine arbeiten, etwas näher betrachtet werden.

4.1 Digital/Analog-Wandler

Das praktisch ausschließlich verwendete technische Konzept für Digital/Analog(D/A)-Wandler ist das der Stromsummation: Am Eingang eines Operationsverstärkers werden Ströme aufsummiert, deren Größe der Wertigkeit der einzelnen Bits in dem umzuwandelnden Digitalwert entspricht. Die schematische Schaltung zeigt Abbildung 36.

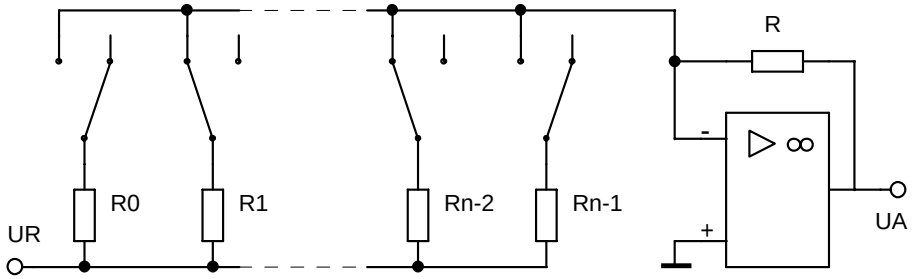


Abbildung 36: Prinzipschaltung eines D/A-Wandlers: Stromsummation.

Beim idealen Operationsverstärker sind Differenzeingangsspannung und Eingangsströme gleich null, daraus ergibt sich für die Ausgangsspannung U_A:

$$U_\mathrm{A} = -U_\mathrm{R} \cdot R \cdot \sum_{i=0}^{n-1} S_i/R_i \tag{4.1}$$

$S_i = 1$, wenn der entsprechende Schalter geschlossen, $S_i = 0$, wenn der entsprechende Schalter geöffnet ist. Die Schalter (meist als Feldeffekttransistoren realisiert) werden

durch die einzelnen Bits des umzusetzenden Digitalwerts angesteuert, die Widerstände R_i haben die Widerstandswerte $R_i = R_0 \cdot 2^{-i}$. Damit erhält man eine Ausgangsspannung, die proportional zum Digitalwert ist.

Die Schaltung Abb. 36 hat den großen Nachteil, dass man sehr unterschiedliche Widerstandswerte benötigt. In der Praxis verwendet man daher eine modifizierte Schaltung, das R/2R-Netzwerk. Das Funktionsprinzip wird aus Abb. 37 deutlich, wenn man sich klarmacht, dass an den Punkten $i = 0, 1, \ldots, n-1$ jeweils die Spannung $U_\mathrm{R} \cdot 2^{-i}$ anliegt.

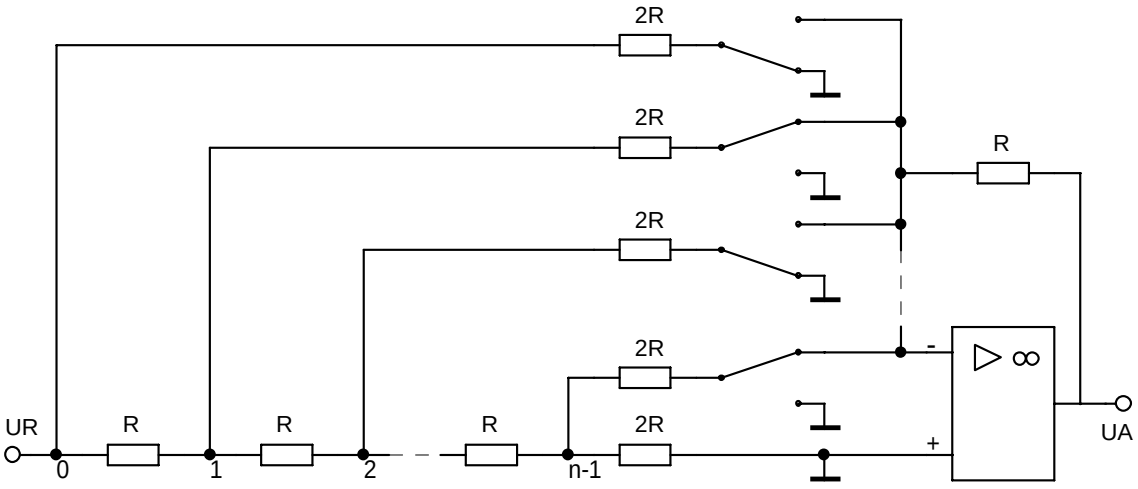


Abbildung 37: Digital/Analog-Wandler mit R/2R-Netzwerk

Die Güte eines D/A-Wandlers und damit seine Verwendungsmöglichkeit für eine bestimmte Steuerungsaufgabe lässt sich durch einige Kenngrößen charakterisieren:

- Die **Auflösung** gibt die Bitbreite des gewandelten Digitalwertes an. Sie liegt im allgemeinen zwischen 8 und 16 Bit, durch die neueren Entwicklungen im Audiobereich (CD-Player) sind zur Zeit auch D/A-Wandler mit hoher Auflösung (14 Bit, 16 Bit) relativ preisgünstig. Die Auflösung bestimmt die minimale Schrittweite, mit der eine Steuergröße vom Rechner vorgegeben werden kann.

- **Genauigkeit** und **Linearität** sowie deren Temperaturabhängigkeit geben an, wie gut die vorhandene (digitale) Auflösung in die Realität der analogen Welt umgesetzt wird. Diese Größen sind von den Toleranzen und dem Temperatur- und Alterungsverhalten des internen Widerstandsnetzwerks abhängig. Hohe Anforderungen hier bedingen hohen Herstellungsaufwand.

- Die **Geschwindigkeit**, oft angegeben als Anstiegszeit (10%–90%) für einen Ausgangsspannungssprung von der minimalen zur maximalen Ausgangsspannung, kann als Kriterium im allgemeinen außer acht gelassen werden, da die Datenausgabe vom PC fast immer langsamer ist und da die typischen Zeitkonstanten der zu steuernden Geräte (Heizung etc.) meist sehr viel größer sind.

Die Ansteuerung von D/A-Wandlern durch den Rechner hängt davon ab, wie der Digitalteil des Wandlers ausgeführt ist. Im einfachsten Fall sind die 'Schalter' durch TTL-kompatible Logiksignale direkt zugänglich, dann muss ein Baustein vorgeschaltet werden, der den Digitalwert zwischenspeichert und auch nach dem Ausgabebefehl statisch am D/A-Wandler anliegen lässt (z. B. ein paralleler E/A-Baustein 8255).

Ist ein internes Speicherregister im D/A-Wandler vorhanden, wird der Digitalwert entweder parallel oder getaktet seriell übergeben. Parallele D/A-Wandler sind ähnlich wie parallele E/A-Bausteine zu sehen: Der Digitalwert wird – in Bytes aufgeteilt – durch Ausgabebefehle übergeben, ein Trigger-Befehl veranlasst die interne Übernahme des kompletten Werts.

Bei seriellen D/A-Wandlern wird der Digitalwert durch einen Daten- und einen Takteingang seriell (bitweise) in ein Schieberegister eingetaktet, die interne Übernahme wird durch einen weiteren Eingang getriggert. Drei Leitungen reichen also zum Anschluss – z. B. an ein Parallelport im PC – aus. Ein Ansteuerprogramm für diese Art von D/A-Wandlern muss den umzuwandelnden Digitalwert in eine Bitfolge umsetzen und diese zusammen mit richtig gesetztem Taktbit und Übernahmebit z. B. über 3 Leitungen einer parallelen Schnittstelle seriell ausgeben.

4.2 Analog/Digital-Wandler

Im Gegensatz zu den D/A-Wandlern werden unterschiedliche technische Konzepte verwendet, die sich in Auflösung, Geschwindigkeit und Aufwand zum Teil beträchtlich unterscheiden.

4.2.1 Parallel-A/D-Wandler

auch als Flash-A/D-Wandler bezeichnet, vergleichen die anliegende Analogspannung in einer Reihe von 2^n Komparatoren[10] mit 2^n Vergleichsspannungen (Abbildung 38), n ist die Bitbreite des Digitalwertes.

Das Prinzip ist sehr aufwendig (für einen 8-Bit-Wandler benötigt man 256 Komparatoren), allerdings auch sehr schnell. Die schnellsten Flash-Wandler erreichen Wandlungsraten im Gigahertzbereich. Eingesetzt werden solche Wandler in Digitalisierern, mit denen sehr schnelle Vorgänge aufgezeichnet werden sollen (Transientenrecorder, Digitalspeicheroszilloskope). Diese Digitalisierer speichern die Daten zunächst intern in einem entsprechend schnellen Speicher, aus dem sie dann langsamer abgerufen werden können. Im PC sind Flash-Wandler zum Beispiel zur Digitalisierung von Videosignalen sinnvoll einsetzbar (Frame-Grabber-Karten), die Geschwindigkeit des direkten Speicherzugriffs (DMA) über

[10] Ein Komparator kann als speziell beschalteter Operationsverstärker angesehen werden, der die beiden Eingangsspannungen vergleicht und das Vergleichsresultat in ein Logiksignal umsetzt. $U_+ > U_- \Rightarrow 1$, $U_+ < U_- \Rightarrow 0$.

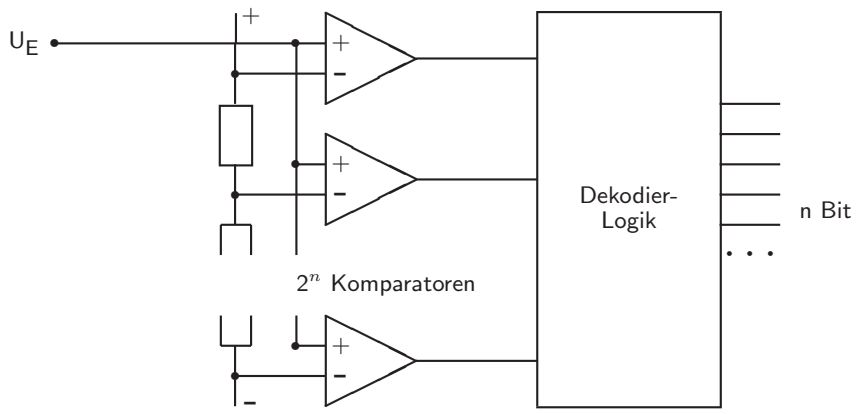


Abbildung 38: Parallel-Analog/Digital-Wandler

den PCI-Bus ist schnell genug, um Videosequenzen ohne Zwischenspeicherung direkt in den Rechnerspeicher zu transferieren.

4.2.2 Kaskaden-Wandler

Um mit der Geschwindigkeit von Parallel-Wandlern eine höhere Auflösung zu erreichen, ohne gleichzeitig den massiven Aufwand von 2^n parallel arbeitenden Komparatoren treiben zu müssen, wurden Kaskadierungskonzepte für Parallel-Wandler entwickelt. Die Eingangsspannung wird bei diesen Wandlertypen in einer ersten Stufe grob digitalisiert, z. B. mit einer Genauigkeit von 6 Bit. In einem schnellen D/A-Wandler wird aus diesem groben Digitalwert eine Kompensationsspannung generiert, die von der Eingangsspannung subtrahiert wird. Die Differenz wird definiert verstärkt und in einem zweiten Digitalisierungsschritt einem weiteren D/A-Wandler zugeführt. Das Gesamtergebnis wird aus den beiden Teilergebnissen zusammengesetzt.

4.2.3 Integrations- und Zählverfahren

Bei diesem Verfahren wird durch die Messspannung der Strom einer Kondensatoraufladung gesteuert (Operationsverstärker als Integrator geschaltet). Die Zeit, die benötigt wird, um eine bestimmte Aufladespannung zu erreichen, ist umgekehrt proportional zum Ladestrom und damit zur Messgröße. Durch Abzählen der Taktimpulse eines hochgenauen Taktgenerators (quarzstabilisiert) kann diese Zeit und damit die Messgröße sehr genau bestimmt werden.

Eine Verbesserung dieses ‘Single-Slope’-Verfahrens ist das ‘Dual-Slope’-Verfahren, dessen Funktionsweise in Abb. 39 dargestellt ist. Die Integration über die Messspannung erfolgt hier für eine fest vorgegebene Zeit $t_1 - t_0$, die durch Abzählen von Z_0 Taktimpulsen gemessen wird. Die nach dieser Zeit erreichte Ladespannung ist proportional zur Messgröße.

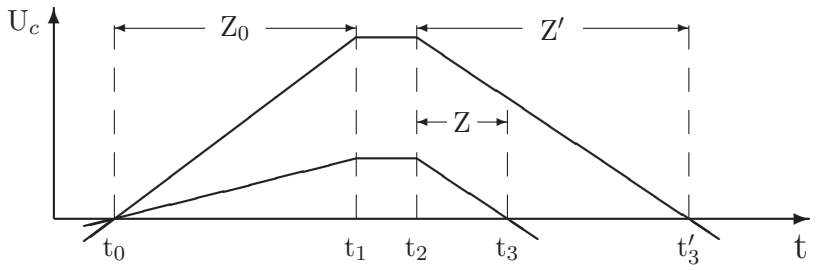


Abbildung 39: Dual-Slope-Verfahren: Zeitlicher Spannungsverlauf an der Integratorkapazität für zwei verschiedene Messspannungen.

Zum Zeitpunkt t_1 wird die Messspannung abgeschaltet, zum Zeitpunkt t_2 stattdessen eine hochgenaue Referenzspannung umgekehrten Vorzeichens am Integrator angelegt. Die Zeit bis zur Entladung des Kondensators $(t_3 - t_2)$ ist nun – da die steuernde Referenzspannung konstant ist und damit die Steigung der Entladegeraden fest – proportional zur Ladespannung bei t_2. Auch diese Zeit wird durch Zählen von Taktimpulsen gemessen (Z, Z'). Die Messspannung kann dann auf sehr einfache Weise berechnet werden:

$$U_{\text{mess}} = -U_{\text{ref}} \cdot Z/Z_0 \tag{4.2}$$

Zweckmäßigerweise werden $-U_{\text{ref}}$ und Z_0 so gewählt, dass ihr Quotient der geforderten Auflösung entsprechen (z. B. $-U_{\text{ref}}/Z_0 = 1\ \mu V$), dann ist Z ohne weitere Umrechnung der Zahlenwert der Messspannung.

Der große Vorteil des Dual-Slope-Verfahrens besteht darin, dass Ungenauigkeiten des Taktgebers und des Integrators nur eine geringe Rolle spielen, da sie in Auflade- und Entladevorgang in gleicher Weise eingehen und damit das Messergebnis in erster Näherung nicht verfälschen. Zur Unterdrückung von Störspannungen kann die Aufladezeit $t_1 - t_0$ so festgesetzt werden, dass sie einem ganzzahligen Vielfachen der Periode der Störspannung entspricht. Üblich sind Vielfache von 20 millisec, um 50 Hz-Störungen auszuschalten.

Integrierende A/D-Wandler sind die weitaus genauesten, das Verfahren wird insbesondere bei Digitalmultimetern eingesetzt. Ein kleiner Nachteil sind die relativ langen Integrationszeiten, die keine allzu schnelle Messfolge zulassen.

Eine etwas modifiziertes Verfahren wird bei der Impulshöhenanalyse in Vielkanalanalysatoren benutzt. Eine Kapazität wird auf eine Spannung aufgeladen, die dem Spitzenwert des zu analysierenden Impulses entspricht, die Entladung erfolgt dann mit konstantem Strom. Die Entladezeit wird wie beim Dual-Slope-Verfahren gemessen, der Zahlenwert ist proportional zur Impulshöhe und dient zur Adressierung des Vielkanalanalysatorspeichers. Hierbei sind Taktfrequenzen von einigen 100 MHz Stand der Technik, die Entladezeit und damit der notwendige Zeitabstand zum nächsten Impuls beträgt etwa 10 μsec.

4.2.4 Wägeverfahren

Beim Wäge- oder Kompensationsverfahren wird die Messspannung in einem Komparator mit einer Vergleichsspannung verglichen, die durch einen Digital/Analog-Wandler erzeugt wird. Die vereinfachte Schaltung zeigt Abb. 40.

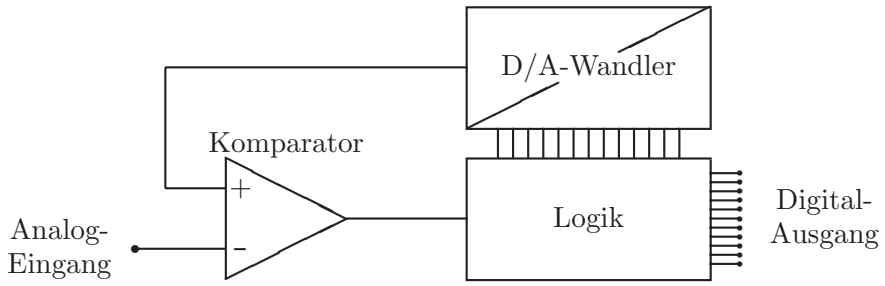


Abbildung 40: Grundschaltung von Analog/Digital-Wandlern, die nach dem Vergleichsprinzip arbeiten.

Im Logikteil wird der Digitalwert der Vergleichsspannung generiert und der Ausgang des Komparators beobachtet. Dieser Teil kann durch Hardware auf dem Baustein oder durch Rechnersoftware realisiert sein. Die Vergleichsspannung kann durch unterschiedliche Strategien variiert werden:

- Bei der einfachen Abzählstrategie wird der Digitalwert, bei 0 beginnend, solange hochgezählt, bis der Komparatorausgang von logisch 0 auf 1 wechselt. Diese Strategie ist die langsamste.
- Beim Nachlaufverfahren wird – abhängig vom Komparatorausgang – vorwärts oder rückwärts gezählt, bis der richtige Wert erreicht ist. Dies Verfahren ist sehr gut geeignet bei langsam veränderlicher Messgröße, in diesem Fall liegt praktisch immer der richtige Digitalwert am Ausgang an.
- Beim Intervallschachtelungsverfahren (sukzessive Approximation) werden, beginnend beim höchstwertigen Bit, alle Bits nacheinander abgetestet und je nach Komparatorreaktion im Digitalwert auf 0 oder 1 gesetzt. Der Messwert wird durch die fortgesetzte Halbierung des Intervalls sehr schnell erreicht. Voraussetzung ist, dass die Messspannung während der Intervallschachtelung einigermaßen konstant bleibt. Dies wird im allgemeinen durch eine vorgesetzte 'Sample-and-Hold'-Schaltung erreicht, die auf ein Startsignal hin den Messwert abtastet (sample) und dann festhält (hold).

Die meisten der auf PC-Karten verwendeten Analog/Digital-Wandler arbeiten nach diesem letztgenannten Verfahren. Ein Programm für solche Wandler muss zunächst eine A/D-Wandlung starten, sodann eine festgelegte Mindestzeit (Wandlungszeit) oder ein 'Ready'-Signal abwarten, dann den Digitalwert einlesen. Die Wandlungszeit liegt, abhängig vom

A/D-Wandler, zwischen 1 und 100 μsec. Häufig ist dem A/D-Wandler ein Analogmultiplexer vorgesetzt, der zwischen mehreren Messstellen umschalten kann, dann muss das Programm zuallererst den richtigen Kanal anwählen.

Die folgenden Beispielfunktionen sollen die drei beschriebenen Strategien noch etwas näher erläutern. Die Funktionen arbeiten mit einem A/D-Wandler zusammen, bei dem der Logikteil durch Rechnersoftware realisiert wird.

Vorab die Definition einer Klasse `CInOut` , in der die Compilerspezifika anonymisiert werden (hardwarenahe Befehle wie die hier für Ein- und Ausgabe benötigten sind nicht standardisiert):

```
typedef unsigned short USHORT;
#ifdef _MSC_VER  // compiler specific
class CInOut {
public:
  void OUTP (USHORT port, int value)
    { _outp (port, value); };
  int INP (USHORT port)
    { return _inp (port); };
};
#else  // different compiler (for M$ eaters)
#endif
```

Die Klassendefinition für `CAdc` enthält die Deklarationen der benötigten Variablen und deren Initialisierung sowie die D/A-Wandler-Prozedur – der 12-Bit-D/A-Wandler wird mit 2 aufeinanderfolgenden Adressen (LowByte⇒daAdr, HighByte⇒daAdr+1) angesprochen:

```
class CAdc : public CInOut {
private:
  USHORT base, adMux, adComp, gtMask, maxAD, ADBits;
  void DA (USHORT daValue)
    { DA (0, daValue); };
  bool IsGreater()
    { return ((INP(adComp) & gtMask) != 0); };
public:
  CAdc (USHORT baseAdr = 0x390) {
    base = baseAdr;
    adMux = base + 8;
    adComp = base + 9;
    gtMask = 1;
    ADBits = 12;
    maxAD = (1<<ADBits) - 1;
  };
  void DA (USHORT daChannel, USHORT daValue) {
```

```
    USHORT daAdr = base + 2*daChannel;
    OUTP (daAdr, daValue);
    OUTP (daAdr+1, (daValue >> 8));
  };
  USHORT AD (int adChannel);
};
```

Die erste Beispielfunktion `AD` veranschaulicht das Abzählprinzip; der Digitalwert wird inkrementiert, bis der Komparator Erfolg meldet oder das Ende des A/D-Bit-Bereichs erreicht ist:

```
USHORT CAdc::AD (int adChannel)
{
  USHORT adValue = 0;
  OUTP (adMux, adChannel);
  while ((adValue<maxAD) && !IsGreater())
    DA (++adValue);
  return adValue;
}
```

Die zweite Variante verwendet das Nachlaufverfahren, es wird – abhängig vom Komparatorausgang – dekrementiert und/oder inkrementiert. Der gemessene Digitalwert muss beim nächsten Aufruf wieder zur Verfügung stehen, daher als statische Variable deklariert werden.

```
USHORT CAdc::AD (int adChannel)
{
  static USHORT adValue;
  OUTP (adMux, adChannel);
  while ((adValue>0) && IsGreater())
    DA (--adValue);
  while ((adValue<maxAD) && !IsGreater())
    DA (++adValue);
  return adValue;
}
```

Beim Verfahren der sukzessiven Approximation werden in einer `for`-Schleife nacheinander die einzelnen Bits abgetestet. Man beginnt beim höchstwertigen, sie werden einzeln zunächst versuchsweise und abhängig vom Komparatorergebnis dann endgültig gesetzt. Nach der Ausgabe des Analogwerts ist eine Wartezeit einzuhalten, die von der Einstellgeschwindigkeit des D/A-Wandlers abhängt.

```
USHORT CAdc::AD (int adChannel)
```

```
{
  USHORT adValue = 0, TestBit;
  OUTP (adMux, adChannel);
  for (int i=ADBits-1; i>=0; i--) {
    TestBit = 1<<i;
    DA (adValue | TestBit);
    // ***** wait for settling *****
    if (!IsGreater())
      adValue |= TestBit;
    }
  return adValue;
}
```

In den obigen Formulierungen liefern die beiden ersten Funktionen einen Digitalwert, der den Analogwert von oben her nähert, die dritte Funktion dagegen die Näherung von unten her. Wo nötig, lässt sich dies leicht ändern.

4.2.5 Spannungs-Frequenz-Wandlung

Bei diesem Verfahren wird die Messspannung in eine Wechselspannung oder eine Impulsfolge umgesetzt, deren Frequenz zur Messgröße proportional ist. Dies leisten spezielle ICs, Spannungs/Frequenz-Wandler. Die Spannungsmessung ist damit in eine Frequenzmessung transformiert. Die Frequenz wird mit einem Zähler gemessen, der auf eine feste Zählzeit eingestellt wird, fortlaufend misst und die jeweiligen Messwerte zum Auslesen in einem 'Latch' zwischenspeichert (Ratemeter).

Der Vorteil des Verfahrens liegt u. a. darin, dass es integrierend ist, damit sehr rauschunempfindlich und wenig störanfällig. Allerdings zählt es zu den langsameren A/D-Wandlungsprinzipien, Zählzeiten von 0.01 – 1 sec sind gebräuchlich. Ein weiterer Vorteil – oft wesentlich – ist die einfach durchzuführende Potentialtrennung zwischen Messstelle und Rechner (nächster Abschnitt).

4.3 Potentialtrennung

Das Arbeiten mit analogen Größen erfordert ein stabiles und vor allem einheitliches Referenzpotential im gesamten Messsystem ('Messerde'). Die Problematik wird deutlich, wenn man sich klarmacht, dass bei 12 Bit Auflösung und einem Spannungsbereich 0...5 V die einem Bit entsprechende Spannung etwa 1 mV beträgt. Störspannungen in dieser Größenordnung lassen sich in einem größeren System oft nicht verhindern. Ein Ausweg, der Störungen meist abhilft, ist die elektrische Trennung von Mess- bzw. Steuerungssystem und Datenverarbeitung. Einheitliche Bezugspotentiale brauchen dann nur noch in den Teilsystemen vorhanden zu sein.

Informationen zwischen den Teilsystemen werden am einfachsten auf optischem Wege übertragen. Dies kann durch Optokoppler[11] oder – bei höheren Anforderungen, z. B. in einer Hochspannungsumgebung – mit Lichtleiterverbindungen realisiert werden. Für die optische Ankopplung sind insbesondere Bausteine und Geräte geeignet, die über wenige Leitungen angebunden werden können, da ansonsten der Aufwand hoch wird. Man verwendet daher für die analoge Ankopplung in einem gestörten Bereich Spannungs/Frequenz-Wandler, die sehr nahe an der Messstelle angeordnet sind und ihre Messfrequenz optisch an einen Zähler im Rechner weitergeben bzw. seriell ansteuerbare D/A-Wandler, die vom Rechner aus optisch angesteuert werden.

4.4 Digitale Regelung

Ein wichtiges Anwendungsgebiet für A/D- und D/A-Wandler ist der Bereich der Regelungstechnik. Das grundlegende Prinzip, nach dem alle Regler – analoge wie digitale – arbeiten, ist das des geschlossenen Regelkreises (Abbildung 41).

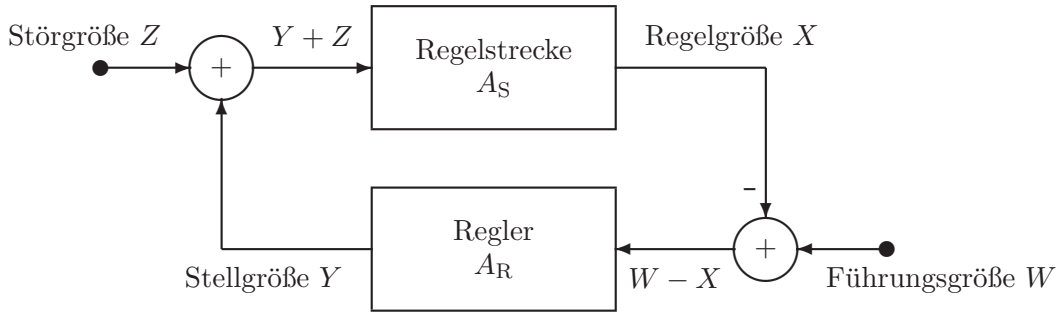


Abbildung 41: Regelkreis

Die **Regelstrecke** ist das Gerät, das geregelt werden soll, z. B. ein Ofen oder Kryostat. Sie wird beschrieben durch ihre Übertragungsfunktion A_S, die im Idealfall eine Konstante ist (lineare Regelstrecke).

Die **Regelgröße** ist der physikalische Parameter, der geregelt werden soll, z. B. die aktuelle Temperatur (Ist-Temperatur, Ist-Wert).

Die **Führungsgröße** ist der Vorgabewert für die Regelgröße (Soll-Temperatur, Soll-Wert).

Der **Regler** ist ein Verstärker im ganz allgemeinen Sinne mit bestimmten Eigenschaften, die durch die Übertragungsfunktion A_R beschrieben werden. Beim **Zwei-Punkt-**

[11] Optokoppler sind Schaltkreise, die eine Lumineszenzdiode und eine Photodiode – elektrisch getrennt, optisch gekoppelt – enthalten. Die Isolationsspannungen liegen im Bereich von einigen hundert bis etwa 2000 Volt.

Regler ist die Übertragungsfunktion eine Stufenfunktion mit Hysterese (Bimetallthermostat o. ä.), beim **Proportionalregler** eine Konstante, die Verstärkung des Reglers.

Die **Stellgröße** ist die physikalische Größe, mit der die Strecke betrieben wird (Heizleistung, -strom, -spannung).

In der **Störgröße** sind die von außen einwirkenden Störungen zusammengefasst (Schwankungen der Umgebungstemperatur, zusätzliche Wärmezufuhr durch Lichteinstrahlung o. ä.).

Idealisiert gilt bei linearer Regelstrecke für einen Proportionalregler

für die Stellgröße

$$Y = A_{\mathrm{R}} \cdot (W - X), \tag{4.3}$$

für die Regelgröße

$$X = A_{\mathrm{S}} \cdot (Y + Z) \tag{4.4}$$

und damit für die Abhängigkeit der Regelgröße von Führungsgröße und Störung

$$X = \frac{A_{\mathrm{R}} A_{\mathrm{S}}}{1 + A_{\mathrm{R}} A_{\mathrm{S}}} \cdot W + \frac{A_{\mathrm{S}}}{1 + A_{\mathrm{R}} A_{\mathrm{S}}} \cdot Z. \tag{4.5}$$

Um den Einfluss der Störung Z gering zu halten, muss man A_{R} sehr groß machen. Dies hat dann Nachteile, wenn Regelstrecke und Regler sich nicht so ideal verhalten wie angenommen. Beispielsweise führen die immer vorhandenen Verzögerungszeiten in Regler und Regelstrecke dazu, dass die Regelung beim Überschreiten einer bestimmten Gesamtverstärkung schwingt. Ein Ausweg ist, die reine Proportionalregelung dadurch zu erweitern, dass man die Vorgeschichte und den Trend mit berücksichtigt. Zum Proportionalteil $A_{\mathrm{R}} \cdot (W - X)$ wird ein Integralteil

$$A_{\mathrm{I}} \cdot \frac{1}{\tau_{\mathrm{I}}} \cdot \int_{-\infty}^{t_0} (W(t) - X(t)) \cdot \exp \frac{t - t_0}{\tau_{\mathrm{I}}} \cdot dt \tag{4.6}$$

und ein Differentialteil

$$A_{\mathrm{D}} \cdot \tau_{\mathrm{D}} \cdot \frac{d(W - X)}{dt} \tag{4.7}$$

hinzugenommen (**PID-Regler**).

PID-Regler sind im analogen Bereich heute Stand der Technik. Ein Analogregler kann als (linearer) Verstärker angesehen werden, bei dem die Verstärkung, sowie Integral- und Differentialanteil variiert werden können.

Beim digitalen Regler wird die gemessene Regelgröße in einen Digitalwert umgesetzt (A/D-Wandler), die Führungsgröße liegt digital als Konstante, Funktion oder Tabelle vor und der eigentliche Regelalgorithmus ist durch ein Programm realisiert. Die Stellgröße wird durch einen D/A-Wandler wieder analog gemacht und – nach entsprechender Verstärkung

– der Regelstrecke zugeführt. Wichtig ist eine sinnvolle Diskretisierung, der Zeittakt dafür kann intern (Timer) oder extern (Messtakt eines Digitalmultimeters) vorgegeben werden.

Der Vorteil des Analogreglers ist sicher im geringeren Aufwand und damit auch geringeren Preis zu sehen; wo dies keine allzu große Rolle spielt, sprechen einige wesentliche Punkte für die Verwendung von digitalen Reglern (Aufzählung ohne Anspruch auf Vollständigkeit):

- Die Führungsgröße kann eine beliebige zeitliche Funktion sein. So lassen sich in Schmelzöfen (Kristallzüchtung) sehr komplexe Temperaturprogramme realisieren.
- Störgrößen (Umgebungstemperatur etc.) können separat gemessen und in der richtigen Weise berücksichtigt werden, bevor ihr Einfluss – zeitlich verzögert – in der Regelgröße zu sehen ist.
- Die Übertragungsfunktion darf auch sehr kompliziert sein, den Rechner stört das nicht. Insbesondere lassen sich Nichtlinearitäten der Regelstrecke berücksichtigen und Grenzbedingungen für die Stellgröße oder deren Änderung festlegen.
- Driftprobleme spielen nur noch in den Wandlern eine Rolle, nicht mehr im Reglerteil.
- Die Übertragungsfunktion kann während der Regelung geändert werden, es lassen sich Regelalgorithmen konstruieren, die sich während der Regelung an die Eigenschaften der Regelstrecke anpassen (adaptive Regler).
- Man kann mit unscharf formulierten Regelalgorithmen arbeiten – *Fuzzy-Regler*.
- Mehrere Stellgrößen und mehrere Regelgrößen lassen sich kombinieren.

5 MATLAB I: Messdatenerfassung

Wenn man einigermaßen komfortable Programme zur Datenerfassung und Steuerung an Experimenten zu erstellen hat, kann das zu einer sehr langwierigen und aufwendigen Aufgabe werden, insbesondere dann, wenn neben der reinen Datenerfassung auch noch weiter gehende Funktionen wie Datenanalyse, Graphikerstellung, Anpassungsrechnungen integriert sein sollen. Allerdings sind die Teilprobleme in vielen Bereichen gleichartig und lassen sich sehr gut modularisieren. Bestimmte Peripheriegeräte sind zu bedienen, Messdaten müssen gepeichert werden, Transformationen oder Fits sind nötig. Das legt es nahe, für die Teilaufgaben fertige Module bereitzustellen, die dann nur noch geeignet kombiniert werden müssen. Verschiedene Hersteller bieten dafür spezialisierte graphische Entwicklungsumgebungen an, mit denen man Mess- und Steuerprogramme relativ einfach bausteinartig zusammensetzen kann. Führend auf diesem Markt sind derzeit die Umgebungen LabView® von National Instruments [16] und VEE® von Agilent [17]. Verwendet man solche Werkzeuge, ist es allenfalls noch notwendig, kurze Treiberroutinen für exotische Hardware im klassischen Sinne selbst zu programmieren. Fast alle kommerziellen Hardwarehersteller liefern zu ihrer Hardware fertige Module für eine oder beide der genannten Entwicklungswerkzeuge mit. Im industriellen Umfeld hat sich diese Art der graphischen Messprogrammerstellung inzwischen weitgehend durchgesetzt, da man damit die Entwicklungszeiten beträchtlich verkürzen kann und da die Kosten der Werkzeuge meist keine allzu große Rolle spielen.

> Effizient zu progammieren heißt auch,
> effizient von fertigen Produkten Gebrauch zu machen.

Ein anderes, in Bereichen wie Physik oder Informatik näher liegendes Konzept zur Arbeitserleichterung ist die Kombination von Numerikprogrammen mit kurzen selbstgeschriebenen Routinen. Die selbstgeschriebenen Teile übernehmen die Kopplung an die Messperipherie, das Numerikprogramme alle weiter gehenden Aufgaben. Dabei ist es sinnvoll, ein Numerikprogramm zu verwenden, das auch auf anderen Gebieten (Theorie, Simulation) gut einsetzbar ist. Die umfangreichsten und modernsten Möglichkeiten bietet hier seit geraumer Zeit MATLAB®, ein Produkt der Firma MathWorks [18]. Ebenfalls recht vielseitig, aber nicht ganz so professionell und benutzerfreundlich ist Scilab [19], ein Programm aus dem *Public-Domain*-Bereich.

Zur Informationsübertragung zwischen den Progammteilen – Daten und Steuerungsanweisungen müssen ausgetauscht werden – können unterschiedliche Mechanismen implementiert sein, betriebssystemspezifische (Pipes, DDE[12], OLE[13], ActiveX, COM[14], DCOM[15], CORBA[16] usw.), aber auch weitgehend betriebssystemunabhänge (Dateien – binär oder

[12] *D*ynamic *D*ata *E*xchange.
[13] *O*bject *L*inking and *E*mbedding.
[14] *C*omponent *O*bject *M*odel.
[15] *D*istributed COM.
[16] *C*ommon *O*bject *R*equest *B*roker *A*rchitecture.

Text – oder direkte Parameterübergabe zwischen den beteiligten Funktionen beipielsweise mit TCP/IP-Mechanismen).

Am Beispiel MATLAB sollen verschiedene Konzepte zur Steuerung und Datenerfassung genauer betrachtet werden. Auf einige der Standardschnittstellen (insbesondere die seriellen Schnittstellen) des Rechners ist ein Zugriff mit MATLAB-eigenen Objekten möglich, ansonsten werden externe Programme oder Funktionen benötigt. Mit der Windows-Version von MATLAB können unter anderem die windowstypischen Kommunikationsstandards DDE und ActiveX verwendet werden, wenn ein geeignetes Partnerprogramm vorhanden ist. So ist beispielsweise ein Datenaustausch mit Programmen wie *Excel* per DDE möglich, mit dem ActiveX-Mechanismus kann Hardware dann angesprochen werden, wenn geeignete Treiber verfügbar sind (Näheres zu beiden Konzepten im MATLAB-Hilfesystem). Wir werden hier die MATLAB-eigenen Standards der MEX-Unterprogramme (*M*ATLAB *EX*tension) und der 'MATLAB Engine' unter Windows näher betrachten, die zwar nicht ganz betriebssystemunabhängig sind, aber in ähnlicher Form auch unter anderen Betriebssystemen (UNIX, Linux) implementiert sind. Das MATLAB-API[17]-Handbuch [20] meint dazu:

> *Although MATLAB is a complete, self-contained environment for programming and manipulating data, it is often useful to interact with data and programs external to the MATLAB environment. MATLAB provides an Application Program Interface (API) to support these external interfaces. The functions supported by the API include:*

- ◇ *Calling C or Fortran programs from MATLAB.*
- ◇ *Importing and exporting data to and from the MATLAB environment.*
- ◇ *Establishing client/server relationships between MATLAB and other software programs.*

MATLAB ist ursprünglich als reines Text-*Frontend* für Numerik-Pakete entwickelt worden, daher ist eine Bedienung über Texteingaben oder Skripte nahe liegend. Es wurden aber auch schon früh Möglichkeiten integriert, graphische Benutzeroberflächen zu erstellen. Dies ist besonders für Anwendungen in der Messdatenerfassung interessant, bei denen sich gleichartige Abläufe (Messungen) häufig wiederholen können. Auch einige dieser Möglichkeiten sollen an Beispielen diskutiert werden.

5.1 Hardware-Zugriff mit MATLAB-Funktionen

Ein Teil der PC-Hardware kann von MATLAB aus direkt angesprochen werden, so die seriellen und die Druckerschnittstellen.

Die Funktion `serial` erstellt ein MATLAB-Objekt für die serielle Schnittstelle, mit der Anweisung

[17] *A*pplication *P*rogram *I*nterface.

```
S1 = serial('COM1', 'BaudRate', 9600);
```

beispielsweise wird die erste serielle Schnittstelle auf eine Geschwindigkeit von 9600 Baud eingestellt und in `S1` bereitgestellt. Weiter verwendet wird `S1` dann ähnlich wie ein Datei-Objekt. Nach `fopen(S1)` können Daten mit `fprintf(S1,...)` ausgegeben, mit `x=fscanf(S1)` gelesen werden. Weitere Schnittstelleneigenschaften werden mit `set(S1,...)` eingestellt, `fclose(S1)`) schließlich beendet die Verbindung.

Ebenfalls wie Datei-Objekte werden die Druckerschnittstellen gehandhabt. Man hat die Wahl zwischen C-artigen und Java-artigen Funktionen[18]:

```
Init = sprintf ('%cU1%c8%cs%c', 27, 27, 27, 1);
Pos  = sprintf ('%cl%c.%c\r\n', 27, Position, 7);
printer = fopen ('lpt1:', 'w');
fprintf (printer, Init);
fprintf (printer, Pos);
fclose (printer);
```

oder

```
ESC = 27; BELL = 7; CR = 13; LF = 10;
Init = [ESC 'U1' ESC '8' ESC 's' 1];
Pos  = [ESC 'l' Position '.' BELL CR LF];
printer = java.io.FileWriter('lpt1:');
printer.write(Init);
printer.write(Pos);
printer.close; .
```

Beide Fragmente positionieren ein Epson-kompatiblen Drucker auf `'Position'`.

Darüber hinaus ist es mit Java-Objekten auch möglich, die Ethernet-Schnittstelle des Rechners von MATLAB aus anzusprechen. Für Netzwerk-Verbindungen ist das *Package* `java.net` zuständig, das explizit importiert werden muss.

Ein MATLAB-Skript für einen Ethernet-Client, der an den Server `ipc1` zum TCPIP-Port `1234` eine Zeichenfolge `sendStr` schickt und die Zeile `recvStr` empfängt, könnte somit etwa diese Anweisungen enthalten:

```
import java.net.*
socket = java.net.Socket('ipc1.physik.uni-osnabrueck.de', 1234);
str = java.lang.String(sendStr);
out = socket.getOutputStream;
out.write(str.getBytes);
in = socket.getInputStream;
```

[18] Die neueren Versionen von MATLAB (in Release 11 inoffiziell und rudimentär, ab Release 12 offiziell) stellen ein Software-Interface zur *Java Virtual Machine* des Rechners bereit, über das einerseits komplette Java-Programme in MATLAB eingebunden werden können, andererseits aber auch einzelne Java-Anweisungen an die *VM* geschickt werden können.

```
isr = java.io.InputStreamReader(in);
ibr = java.io.BufferedReader(isr);
recvStr = ibr.readLine;
socket.close; .
```

Die Verbindung wird mit dem `Socket`-Objekt etabliert, mit der `write`-Methode des zugehörigen `OutputStream` wird die Anfrage abgeschickt. Die Antwort wird mit dem `InputStream` gelesen, dem ein `BufferedReader` angehängt wurde, um bequem mit `readLine` lesen zu können.

5.2 Externe Programme

Sollen – von MATLAB aus gesehen – nur einfache Aktionen angestoßen werden, ohne dass Informationen zurückgeliefert werden, kann man das über ein externes Programm erledigen, dem die benötigten Parameter in der Kommandozeile übergeben werden. Ein C++-Programm `epson`, das etwa folgende `main`-Funktion enthält

```
int main(int argc, char* argv[]) {
  pos = atoi(argv[1]);
  CEpson fx;
  fx.Position(pos);
  ... ,
```

könnte aus MATLAB mit

```
Position = 40;
eval(['!epson ' num2str(Position)]);
```

aufgerufen werden (`eval` ermöglicht variable Parameter beim Aufruf eines externen Programms mit '`!`').

5.3 MEX-Funktionen

Von MATLAB aufrufbare C/C++- oder Fortran-Subroutinen (MEX-Dateien) werden unter Windows als DLLs (*D*ynamic *L*ink *L*ibraries) erstellt und beim Aufruf aus MATLAB vom Betriebssystem automatisch geladen und ausgeführt:

> *You can call your own C or Fortran subroutines from MATLAB as if they were built-in functions. MATLAB callable C and Fortran programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute. MEX-files have several applications:*
>
> ◇ *Large pre-existing C and Fortran programs can be called from MATLAB without having to be rewritten as M-files.*

- ◇ *Bottleneck computations (usually for-loops) that do not run fast enough in MATLAB can be recoded in C or Fortran for efficiency.*

Der dem MATLAB-Handbuch [20] entnommenen Auflistung wäre noch der Bereich der Kopplung ans Experiment hinzuzufügen.

Zur Realisierung von MEX-Dateien ist im MATLAB-System das Kommando *mex* vorgesehen; nach richtiger Konfiguration mit `mex -setup` ist es in der Lage, aus C- oder C++-Dateien die DLLs zur Verwendung unter MATLAB zu erstellen. Die Quelldatei wird dazu im jeweils aktuellen Verzeichnis – z. B. mit dem MATLAB-Texteditor – angelegt. Voraussetzung ist, dass auf dem Rechner einer der von MATLAB unterstützten C/C++-Compiler vorhanden ist. Ein einfacher frei verfügbarer C-Compiler (LCC) wird bei MATLAB mitgeliefert, damit können in jedem Fall MEX-Programme compiliert werden (allerdings nur in C).

MATLAB kommuniziert mit einer MEX-Datei über die Interface-Funktion `mexFunction()`, die mithin in jeder MEX-Datei vorhanden sein muss. Diese Funktion hat die Signatur

```
void mexFunction ( int nlhs, mxArray *plhs[],
                   int nrhs, const mxArray *prhs[] ),
```

die 4 Parameter beschreiben die Eingabe- und Ausgabeobjekte der MEX-Funktion (*l*eft *h*and *s*ide, *r*ight *h*and *s*ide). Die Felder `prhs` und `plhs` enthalten Zeiger auf die Objekte, `nrhs` und `nlhs` sind deren Anzahl. Bei einer MEX-Funktion `func.dll`, die aus MATLAB mit

```
[u, v] = func (a);
```

aufgerufen wird, ist `nlhs = 2`, `plhs[0]` zeigt auf `u`, `plhs[1]` auf `v`, `nrhs = 1`, `prhs[0]` zeigt auf `a`.

Der C++-Quellcode einer MEX-DLL, die als einziges Objekt ein zweidimensionales Messdatenfeld zurückgibt, enthält etwa die folgenden Zeilen:

```
#include "mex.h"
...
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
    plhs[0] = mxCreateDoubleMatrix(YSIZE, XSIZE, mxREAL);
    double * z = mxGetPr(plhs[0]);
    ...
    // put data into array z
}  .
```

Das Matrix-Objekt wird mit `mxCreateDoubleMatrix()` angelegt, `plhs[0]` zeigt darauf. Mit `mxGetPr(plhs[0])` beschafft man sich im Programm einen Zeiger auf den Anfang des eigentlichen Datenfelds.

5.4 MEX-Funktionen und 'Microsoft Foundation Classes'

Für kurze MEX-Programme oder solche, die Numerik-Aufgaben erledigen, reicht die beschriebene Vorgehensweise aus. Will man jedoch innerhalb der MEX-Funktion mit einer Benutzeroberfläche arbeiten (Dialog o. ä.), ist es wünschenswert, die DLL innerhalb der gewohnten Umgebung und mit deren Hilfsmitteln zu entwickeln. Dazu ist im Falle der Microsoft-Entwicklungsumgebung für Visual C++ wie folgt vorzugehen:

- Als Projekttyp im Startfenster wird *MFC AppWizard (dll)* angewählt, im Folgefenster *Regular DLL using shared MFC DLL* markiert. Die Entwicklungsumgebung erstellt daraufhin ein Gerüst für eine DLL.
- In der .def-Datei des Projekt wird unter `EXPORTS` der Name der zu exportierenden Funktion `mexFunction` vermerkt.
- Mit
  ```
  #include "mex.h"
  ```
  und dem 'Additional Include Directory:'
  ```
  <MATLABPATH>\extern\include
  ```
  werden dem Programm die nötigen Definitionen zugänglich gemacht.
- Die Bibliotheken `libmx.lib`, `libmex.lib`, `libmatlbmx.lib` und `libmat.lib` werden als zusätzliche Input-Dateien bei den Linker-Optionen angegeben, als zusätzlicher Linker-Suchpfad
  ```
  <MATLABPATH>\extern\lib\win32\microsoft\msvc60
  ```
  (für andere von MATLAB unterstützte Compiler gegebenenfalls anzupassen).

Damit sind die Vorarbeiten erledigt, die MEX-Funktion kann implementiert werden:

```
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  AFX_MANAGE_STATE(AfxGetStaticModuleState());
  plhs[0] = mxCreateDoubleMatrix(CData::YSIZE, CData::XSIZE, mxREAL);
  double * z = mxGetPr(plhs[0]);
  CData DataDlg;
  DataDlg.DoModal();
  for ( int k=0; k<CData::YSIZE; k++ )
    for ( int i=0; i<CData::XSIZE; i++ )
      * (z + i*CData::YSIZE + k) = DataDlg.Z[k][i];
}
```

Wichtig ist das Makro `AFX_MANAGE_STATE...` in der ersten Funktionszeile (Hinweis dazu in der .CPP-Hauptdatei des Projekts). Die `mexFunction()` ruft hier einen modalen

Dialog auf, der die Datenerfassung, gegebenenfalls auch vom Benutzer vorzunehmende Einstellungen des Experiments, realisiert. Am Ende der Funktion werden die Daten ins Rückgabeobjekt umkopiert.

5.5 MEX-Funktionen mit GCC oder G++

Will man MEX-Funktionen mit Gnu-Werkzeugen erstellen, dann gibt es dafür mehrere Möglichkeiten, allerdings (noch) keine direkte Unterstützung von The Mathworks. Für die Gnu-Werkzeuge gibt es zurzeit mindestens drei interessante Windows-Implementierungen:

Cygwin bildet eine komplette UNIX-Umgebung unter Windows nach [21].

UWIN von AT&T verwendet Microsoft-Compiler als Standard, kann aber auch Gnu-Compiler einbinden [22].

MinGW kombiniert die Windows-Bibliotheken mit den Gnu-Werkzeugen [23]. Eine Entwicklungsumgebung dafür ist Dev-C++ [24].

Für die Kombination von Cygwin mit MATLAB gibt es eine ausführliche Beschreibung sowie die notwendigen zusätzlichen Dateien im Netz [25]. Die Implementierung ist analog zum üblichen `mex`-Kommando in MATLAB, MEX-DLLs werden innerhalb der MATLAB-Umgebung erstellt. Voraussetzung ist allerdings die Installation des relativ aufwendigen Cygwin-Systems.

Will man mit dem erheblich schlankeren MinGW und Dev-C++ als Entwicklungsumgebung arbeiten, kann man etwa wie folgt vorgehen:

- Zunächst muss eine Bibliotheksdatei (.LIB) erstellt werden, die alle in MEX-Dateien notwendigen Funktionen enthält. Das erledigt der Aufruf

  ```
  dlltool --def <MATLABPATH>/extern/include/matlab.def \
          --output-lib libmex.lib
  ```

  (in einer Zeile, die Fortsetzung mit '\' funktioniert unter Windows nicht), `dlltool` ist im Gnu-Paket enthalten.

- In Dev-C++ wird mit einem leeren Projekt begonnen, als Projekt-Optionen wird die Erstellung einer DLL gewählt, `libmex.lib` wird als zusätzliche Objekt-Datei angegeben, als zusätzliches Include-Verzeichnis wird

  ```
  <MATLABPATH>/extern/include
  ```

  eingetragen.

- Das wäre schon alles — wenn Dev-C++ fehlerfrei wäre. Leider wird die Lib-Datei nicht berücksichtigt, wenn man aus der Entwicklungsumgebung compiliert und linkt.

Richtig gemacht wird aber der Makefile[19]. Folglich verwendet man die Entwicklungsumgebung nur als Editor, lässt sich einen Makefile generieren und übersetzt die DLL mit `make` aus einem DOS-Fenster oder mit `!make` vom MATLAB-Befehlsprompt.

Um alles, insbesondere auch die Parameterübergabe, zu testen, sollte man mit einer einfachen DLL anfangen, etwa:

```
#include "mex.h"
void mexFunction( int nlhs, mxArray *plhs[],
             int nrhs, const mxArray *prhs[] )
{
  if (nrhs==0)
    return;
  double * y = mxGetPr(prhs[0]);
  int M = mxGetM(prhs[0]);
  int N = mxGetN(prhs[0]);
  plhs[0] = mxCreateDoubleMatrix(M, N, mxREAL);
  double * z = mxGetPr(plhs[0]);
  for (int i=0; i<M; i++)
    for (int k=0; k<N; k++)
      *(z+N*i+k) = *(y+N*i+k)**(y+N*i+k);
}
```

Das Beispiel quadriert eine Matrix feldweise, entspricht mithin der MATLAB-Anweisung `M.*M`.

5.6 MATLAB als 'Engine'

MEX-Funktionen sind Erweiterungen des MATLAB-Befehlsumfangs und werden aus MATLAB aufgerufen. Umgekehrt kann man auch den Befehlsumfang des eigenen Programms durch MATLAB erweitern, MATLAB-Funktionen z. B. aus einem Messprogramm aufrufen:

> *MATLAB provides a set of routines that allows you to call MATLAB from your own programs, thereby employing MATLAB as a computation engine. MATLAB engine programs are C or Fortran programs that communicate with a separate MATLAB process via pipes (in UNIX) and through ActiveX on Windows. There is a library of functions provided with MATLAB that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB. Some of the things you can do with the MATLAB engine are:*

[19] Bei größeren Projekten sollte man den Makefile nachbessern, da derzeit darin – wie auch in der Entwicklungsumgebung – jeweils alle Dateien des Projekts frisch übersetzt werden.

- ◇ *Call a math routine to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.*
- ◇ *Build an entire system for a specific task, for example, radar signature analysis or gas chromatography, where the front end (GUI) is programmed in C and the back end (analysis) is programmed in MATLAB, thereby shortening development time.*

Die Nutzung von MATLAB als 'Engine' veranschaulicht die Funktion `OnFilter()`, die MATLAB aufruft, um ein zweidimensionales Datenfeld (quadratisches Bild `b` der Größe `BSIZE*BSIZE`) zu filtern:

```
#include "engine.h"
...
void CMulDoc::OnFilter()
{
  int i, k;
  Engine * ep = engOpen ("");
  engEvalString (ep, "f=ones(5);");
  mxArray * z = mxCreateDoubleMatrix (BSIZE, BSIZE, mxREAL);
  mxSetName (z, "z");
  double * ptrz = mxGetPr (z);
  for ( i=0; i<BSIZE; i++ )
    for ( k=0; k<BSIZE; k++ )
      * (ptrz + i*BSIZE + k) = b[k][i];
  engPutArray (ep, z);
  engEvalString (ep, "z=filter2(f,z);");
  engEvalString (ep, "z=z-min(min(z));");
  engEvalString (ep, "z=z*255/max(max(z));");
  mxArray * res = engGetArray (ep, "z");
  ptrz = mxGetPr (res);
  for ( i=0; i<BSIZE; i++ )
    for ( k=0; k<BSIZE; k++ )
      b[k][i] = (BYTE) (* (ptrz + i*BSIZE + k));
  mxDestroyArray (res);
  mxDestroyArray (z);
  engClose (ep);
  UpdateAllViews (NULL);
}  .
```

MATLAB wird mit `engOpen()` gestartet, mit `f=ones(5)` wird in MATLAB ein einfaches zweidimensionales Rechteckfilter definiert. Das Datenfeld wird an MATLAB übergeben, mit der zweidimensionalen Filterfunktion `filter2()` gefiltert, wieder auf 8-Bit-Dynamik normiert und zurückkopiert. Lässt man die `engClose`-Anweisung am Ende weg, kann man nach dem Motto 'MATLAB, übernehmen Sie!' mit den übergebenen Daten in MATLAB

weiterarbeiten.

Auch bei Engine-Anwendungen sind gewisse Vorarbeiten nötig: ähnlich wie bei den MEX-Anwendungen müssen zusätzliche Bibliotheken eingebunden werden, diesmal `libmx.lib`, `libmat.lib`, `libeng.lib` (Vorgehensweise für unterstützte Compiler wie Visual C++ analog zu 5.4, für nicht unterstützte Compiler wie G++ müssen die Lib-Dateien zunächst erstellt werden).

Einen ausführlichen und vollständigen Überblick über die von MATLAB bereitgestellten Schnittstellenfunktionen geben die zuständigen Handbücher (Benutzerhandbuch gedruckt [20] und als PDF-Datei [26]; Referenzhandbuch in HTML [27] und PDF [28]) oder das MATLAB-Hilfesystem. Ein weiteres Engine-Beispiel ist im Skriptum zur Vorlesung 'Graphik-Workshop' [29] beschrieben. Dort geht es um den Tranfer von Bilddaten, die aus einer Twain-Quelle stammen, an MATLAB.

5.7 ActiveX

In der Windows-Version kann Matlab ActiveX-Komponenten[20] verwenden, sowohl so genannte *Controls* wie auch *Server*. Damit wird es sehr einfach, mit Rechnerperipherie zu arbeiten, für die ActiveX-Controls vom Hersteller mitgeliefert werden. Dies ist inzwischen bei sehr vielen kommerziellen Steckkarten und auch bei externen Geräten zur Datenerfassung und Steuerung der Fall. Auch bei selbst programmierten Software-Komponenten sollte man die Möglichkeit in Betracht ziehen, den ActiveX-Standard zu benutzen. Dies zumindest dann, wenn man mit einem Software-Entwicklungssystem arbeitet, das dies einigermaßen einfach ermöglicht.

Matlab-Objekte für ActiveX-Controls werden mit

```
h = actxcontrol('Programm-ID',...);
```

erstellt. `Programm-ID` ist der Name, unter dem die ActiveX-Komponente in der *Registry* eingetragen ist. Als weitere Parameter können beim Aufruf angegeben werden (s. Matlab-Hilfe): Position und Größe des Controls, Handle zum *Parent*-Objekt und eine oder mehrere *Callback*-Funktionen für Ereignisse (Events) des Controls. Die möglichen Ereignisse können mit `events(h)` erfragt werden.

Eigenschaften können, wie bei allen Matlab-Objekten üblich, mit `set` und `get` festgelegt und abgefragt werden, interaktiv geht das einfacher mit dem *Property Editor*, der mit `inspect(h)` aufgerufen wird.

Methoden – Funktionen – des Objekts ruft man entweder mit

```
a = invoke(h, 'Function', Parameter1, Parameter2, ...);
```

[20] ActiveX ist eine der innerhalb des Microsoft Component Object Model (COM) definierten Software-Schnittstellen. Software-Komponenten – in der Regel DLLs – unterschiedlicher Herkunft können über diese Interface-Definition relativ problemlos zu variablen größeren Systemen zusammengefasst werden.

oder mit

```
a = Function(h, Parameter1, Parameter2, ...);
```

auf. Ein leeres `invoke(h)` liefert eine Liste der mit dem Objekt verbundenen Methoden.

5.8 Dateiformate

MATLAB kann eine ganze Anzahl von Dateiformaten direkt erstellen und lesen, daneben sind verschiedene Funktionen implementiert, mit denen die Ein- und Ausgabe sehr flexibel gesteuert werden kann. Reicht dies alles nicht aus, so bleibt noch die Möglichkeit, eigene Funktionen in C/C++, Fortran oder Java zu implementieren.

SAVE und LOAD Die Standardfunktionen zum Schreiben und Lesen von Daten sind `save` und `load`. Ohne Argumente speichert `save` alle Variablen der aktuellen MATLAB-Sitzung binär in der Datei `matlab.mat`, mit `load` wird diese Datei wieder gelesen, die Variablen werden wiederhergestellt. Durch Argumente kann man bei `save` den Namen der Datei, die zu speichernden Variablen, die Formatierung und die Datengenauigkeit vorgeben. So werden mit

```
save  <fname>  X Y Z
```

die Variablen (Matrizen) `X`, `Y` und `Z` in der Datei `<fname>` im MATLAB-eigenen Datenformat gespeichert, mit

```
save  <fname>  Z  -ASCII  -DOUBLE
```

wird MATLAB angewiesen, `Z` in Textformat mit hoher Genauigkeit (16 Nachkommastellen) zu speichern. Bei Textdateien (Option `-ASCII`) ist es meist nicht sinnvoll, mehrere Datenfelder anzugeben, da die Matrizen nacheinander geschrieben werden, ohne dass die Größe und der Variablenname vermerkt werden. Dies kann beim Wiedereinlesen Probleme machen, insbesondere dann, wenn die Spaltenanzahl der Matrizen unterschiedlich ist. Will man – wie bei Messdaten üblich – als Spaltenvektoren vorliegende Messwerte tabellenartig schreiben, muss man sie in MATLAB vorher zu einer Matrix zusammensetzen (`Z = [X,Y]`).

In ähnliche Weise lassen sich auch bei `load` nähere Spezifikationen angeben.

```
load  <fname>
```

liest die Datei `<fname>`. Ohne Dateinamenerweiterung oder mit `.MAT` als Erweiterung geht MATLAB von binärem MATLAB-Format aus, ansonsten von Textformat. Bei Binärdateien werden die Daten unter ihren ursprünglichen Namen wiederhergestellt, durch Angabe von einzelnen Variablennamen als Parameter kann man `load` darauf beschränken. Handelt es sich um eine Textdatei, wird der Inhalt in der Matrix `<fname>` abgelegt. Durch Verwendung der funktionalen Form kann man das verändern. So können durch Leerzeichen getrennte Zahlen aus einer Textdatei `test.dat` mit

```
Z = load('test.dat')
```

in die Matrix `Z` gelesen werden. Deren Spalten- und Zeilenzahl entspricht der Anordnung der Daten in der Textdatei.

Weitere Formate Ein Überblick über weitere Dateiformate, die in MATLAB zum Austausch von Messdaten mit anderen Programmen verwendet werden können, ist im Skriptum zur Vorlesung 'Graphik-Workshop' [29] enthalten. Informationen dazu findet man auch in der MATLAB-Hilfe mit `help fileformats` und `help iofun`.

5.9 Graphische Benutzeroberflächen

Die Bedienung von (Mess-)Programmen wird dadurch beträchtlich erleichtert, dass man für bestimmte Aktionen Eingabehilfen wie Drucktasten oder Menüs zur Verfügung hat. MATLAB bietet zur Erstellung solcher graphischer Oberflächen umfangreiche Hilfsmittel. Basis ist immer ein `figure`-Objekt, das über die Funktion `uimenu` mit zusätzlichen Menüpunkten und über die Funktion `uicontrol` mit Tasten, Eingabefenstern, Auswahllisten und ähnlichem ausgestattet werden kann.

Abbildung 42 zeigt ein einfaches Beispiel, in dem mathematische Funktionen mit den Einfachkommandos `ezplot`, `ezsurf` oder `ezmesh` gezeichnet werden können. Die gewünschte Funktion wird im Textfeld eingegeben, nach Drücken der jeweiligen Taste geplottet.

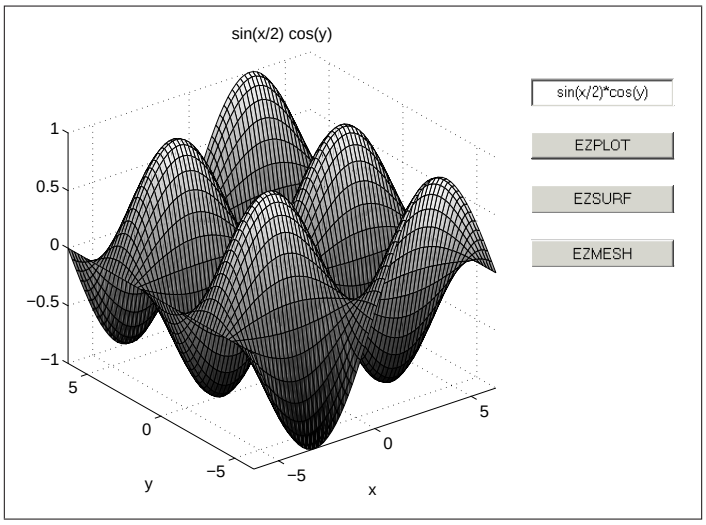


Abbildung 42: Graphische Benutzeroberfläche zum Test von Funktionsplots.

Das zugehörige MATLAB-Skript erstellt zunächst ein `figure`-Objekt als Rahmen, darin mit `axes` eine Zeichenfläche.

```
figure( ...
    'Name','User Interface in MATLAB', ...
    'NumberTitle','off');
```

```
axes( ...
    'Units','normalized', ...
    'Position',[0.10 0.12 0.6 0.78], ...
    'FontUnits','normalized','FontSize',0.055);
Pos = [0.75 0.8 0.2 0.05];
dPos = [0 -0.1 0 0];
hText = uicontrol( ...
    'Style','edit', ...
    'Units','normalized', ...
    'Position',Pos, ...
    'BackgroundColor',[1 1 1], ...
    'String','sin(x/2)*cos(y)');
uicontrol( ...
    'Style','pushbutton', ...
    'Units','normalized', ...
    'Position',Pos+dPos, ...
    'String','EZPLOT', ...
    'Callback','ezplot(get(hText,''String''))');
...
```

Das Textfeld wird vorbesetzt, beim Drücken der Taste (hier `EZPLOT`) wird das Textfeld gelesen und die gewünschte Funktion (`ezplot`) damit aufgerufen.

Die Funktion `uicontrol` erstellt Graphik-Objekte, die durch Eigenschafts-Werte-Paare näher definiert werden. So gibt der Wert für `Style` an, was für ein Typ `uicontrol` erstellt werden soll, `Pos` die Lage und Größe, `String` die Beschriftung. Aktionselemente brauchen eine `Callback`-Eigenschaft, die die auszuführende Aktion festlegt. Der Rückgabewert von `uicontrol` ist ein Objekt-Pointer (*Handle*), mit dem Eigenschaften des Objekts später erfragt (`get`) oder verändert (`set`) werden können.

Das zweite Beispiel implementiert eine einfache Messumgebung (Abbildung 43). Für Messparameter sind Eingabefelder vorgesehen, verschiedene Aktionen können per Tastendruck gestartet werden.

Die Aktionen sind jeweils durch die `Callback`-Eigenschaft festgelegt:

```
uicontrol( 'Style', 'pushbutton', ...
    'String', 'MEASURE', ...
    'Callback', measurecb);
uicontrol( 'Style', 'pushbutton', ...
    'String', 'CALCULATE', ...
    'Callback', 'errordlg(''Not implemented'',''Error'')');
uicontrol( 'Style', 'pushbutton', ...
    'String', 'SAVE', ...
    'Callback', savecb);
```

Beim Drücken der `CALCULATE`-Taste wird mit `errordlg` ein Dialogfenster mit Fehlermel-

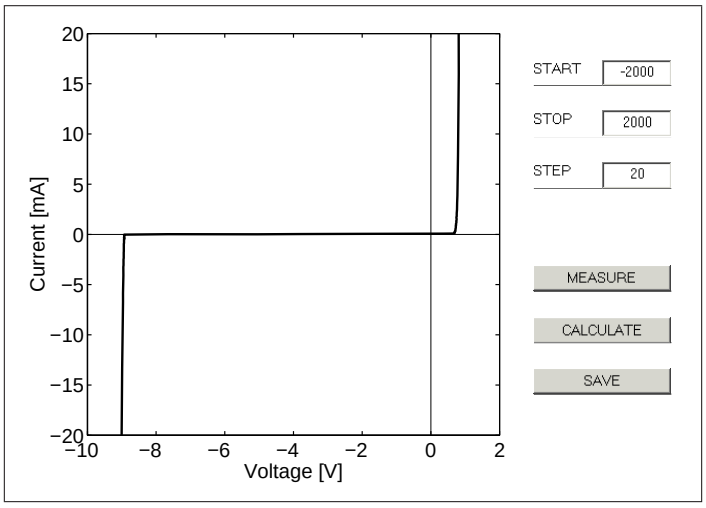


Abbildung 43: Graphische Benutzeroberfläche zur Messdatenerfassung und -darstellung (gemessen wurde gerade die Kennlinie einer 9-V-Zenerdiode).

dung aufgemacht, die beiden weiteren Aktionen sind an anderer Stelle im MATLAB-Skript implementiert, `measurecb` als

```
measuring = 0;
measurecb = [...
    'if measuring==0,' ...
    'measuring = 1;' ...
    'x0 = str2num(get(hStart,''String''));' ...
    'x1 = str2num(get(hStop,''String''));' ...
    'dx = str2num(get(hStep,''String''));' ...
    '[x,y] = domeas(x0,x1,dx);' ...
    'measuring = 0;' ...
    'end;']; .
```

Die Zahlenwerte (`str2num`) der Texteingaben (`hStart`, `hStop`, `hStep` sind die Handles der Textfelder) werden als Parameter an die eigentliche Messfunktion `domeas` übergeben, zurückgeliefert werden die Messwerte. Die *Semaphore* (Ampel) `measuring` sorgt dafür, dass ein zweiter Tastendruck während einer Messung ohne Folgen bleibt. Falls die Messung länger dauert, sollten innerhalb der Messfunktion regelmäßig aktuelle Messdaten gezeichnet werden, um über den Messablauf zu informieren. Dazu wird die Graphik zunächst konfektioniert (Datenbereich, Achsenbeschriftungen usw.) und mit `hold on` eingefroren. In der Messschleife innerhalb der Messfunktion `domeas` wird dann laufend nur noch der aktuelle Datenpunkt geplottet:

```
for n = 1:nmax,
  ...            % Slow Measurement
  plot(x(n), y(n), 'ko');
  if mod(n,10)==0, drawnow; end;
end; .
```

Der Aufruf von `drawnow` sorgt dafür, dass konkret gezeichnet wird, im obigen Fall nach

jedem zehnten Punkt. MATLAB schiebt ansonsten alle Zeichenaktionen auf, bis nichts anderes mehr anliegt, würde also ohne diese Zeile erst am Ende der Messung zeichnen.

Die Aktion `savecb` macht zunächst mit `uiputfile` ein Standarddialogfenster zur Auswahl eines Dateinamens auf, Rückgabewerte sind Dateiname und vollständiger Pfad:

```
savecb = ...
    ['[fn,fp]=uiputfile(''*.*'');', ...
    'if fn~=0,', ...
    '  chdir(fp);', ...
    '  pairs=[x'',y''];' ...
    '  save(fn,''pairs'',''-ascii'');' ...
    'end;']; .
```

Vor dem Abspeichern mit `save` werden die im Messprogramm als Zeilenvektoren vorliegenden Messwerte in die gewünschte Tabellenform gebracht (zweispaltige Matrix `pairs`).

GUIDE Als Hilfsmittel zur graphischen Erstellung von GUIs enthält MATLAB einen graphischen Editor (GUI Design Environment – GUIDE), der mit `guide` aufgerufen wird. Der Editor erstellt eine MATLAB-Graphik-Datei (`.fig`) und eine dazu korrespondierende Skript-Datei (`.m`). Seit Release 13 (Juni 2002) ist es auch möglich, alles in eine einzige Skript-Datei zu exportieren. Damit sind nachträgliche Änderungen und Anpassungen sowohl mit dem graphischen Editor wie auch mit einem Text-Editor möglich. Die Dateien sind allerdings etwas umfangreicher und unübersichtlicher als Skript-Dateien, in denen ein GUI direkt programmiert wird.

6 MATLAB II : Messdatenverarbeitung

MATLAB ist die Abkürzung für *MAT*rix *LAB*oratory, Vektoren und Matrizen sind die Variablentypen, mit denen MATLAB am besten umgehen kann. Das sind aber auch genau die Datentypen, in denen Messdaten üblicherweise vorliegen. Meist eindimensional, eine physikalische Größe wird in ihrer Abhängigkeit von einer anderen, unabhängig veränderlichen gemessen. So wird beim Freien Fall der zurückgelegte Weg in Abhängigkeit von der Zeit bestimmt, die Kennlinie einer Diode ist der Strom als Funktion der anliegender Spannung, die optische Absorption eines Materials wird meist wellenlängenabhängig gemessen, in der Kernspektroskopie interessiert man sich für die Energieverteilung der bei Zerfalls- oder Wechselwirkungsprozessen involvierten Teilchen. Zwei- oder mehrdimensional dagegen liegen Daten beispielsweise überall dort vor, wo physikalische Größen ortsabhängig gemessen werden. Alle bildgebenden Messverfahren – wie die Rastermikroskopie oder auch viele der Untersuchungsmethoden aus dem medizinischen Bereich – liefern solche Daten. Da in MATLAB Rechenverfahren für Matrizen sehr effizient implementiert sind, ist es gerade auch für die Weiterverarbeitung von ein- und mehrdimensionalen Messdaten hervorragend geeignet[21].

> Die in Messdaten enthaltene physikalische Information
> wird oft erst nach geschickter Bearbeitung sichtbar.

6.1 Filterung

Filterverfahren werden verwendet, um Datenfelder zu glätten, aber auch, um spezielle Informationen hervorzuheben. MATLAB stellt dafür verschiedene Funktionen zur Verfügung, die wichtigsten sind `filter` zur Filterung eines eindimensionalen Felds und `filter2` für zweidimensionale Felder. Die Verfahren berechnen die Faltung aus zwei Vektoren oder Matrizen, dem Datenfeld D und der Filterfunktion F.

So berechnet `Df = filter(F,norm,D)` an jedem Punkt des Ergebnis-Feldes `Df` die mit `F` gewichtete und mit `norm` normierte Summe der Werte des Ausgangsfeldes `D`, die in der Umgebung dieses Punktes liegen

$$Df(n) = \frac{D(n)F(1) + D(n-1)F(2) + \ldots + D(n-m+1)F(m)}{\text{norm}} \tag{6.1}$$

$$= \sum_{k=1}^{m} \frac{D(n-k+1)F(k)}{\text{norm}} \,. \tag{6.2}$$

Den Mittelwert aus jeweils 5 benachbarten Datenpunkten würde man demnach durch die Anwendung der Filterfunktion `F = ones(1,5)` und `norm = 5` bzw. `norm = sum(F)` erhalten.

[21] Für bestimmte Anwendungsbereiche bietet The MathWorks spezialisierte *Toolboxes* zu MATLAB an, so die 'Image Processing Toolbox' mit vielen Funktionen für die digitale Bildbearbeitung.

6.1.1 Gewichteter Mittelwert

Die oben beschriebene einfache Mittelwertbildung entspricht einer rechteckigen Filterfunktion. Oft ist es zweckmäßig, einen geeignet gewichteten Mittelwert zu berechnen, um nahe liegende Punkte stärker zu bewerten als entfernte. Die Form und Breite der konkreten Filterfunktion muss jeweils an die gewünschte Anwendung angepasst werden. In den meisten Fällen ist es sinnvoll, symmetrische Funktionen zu verwenden, die an ihren Rändern einigermaßen stetig verlaufen (Trapez, Dreieck, Cosinus o. ä.).

Das folgende Fragment wendet eine parabelförmige Filterfunktion `weight` auf ein verrauschtes Signal an:

```
HalfWidth = 20;
linear = [-HalfWidth:HalfWidth]/HalfWidth;
weight = 1 - linear.*linear;
Filtered = filter(weight, sum(weight), Signal); .
```

Im zweiten Parameter von `filter` kann ein zusätzlicher Wertevektor angegeben werden, mit der vorhergehende Werte des Ergebnisvektors berücksichtigt werden können (Näheres dazu in der Online-Hilfe). Im obigen Fall wird nur eine Konstante zur Normierung eingesetzt.

Die Wirkung zweier unterschiedlich breiter Filterfunktionen zeigt Abbildung 44. Die verwendeten Filterfunktionen sind mit eingezeichnet; deutlich ist die Abhängigkeit der Glättungswirkung, aber auch des Informationsverlusts von der Filterbreite.

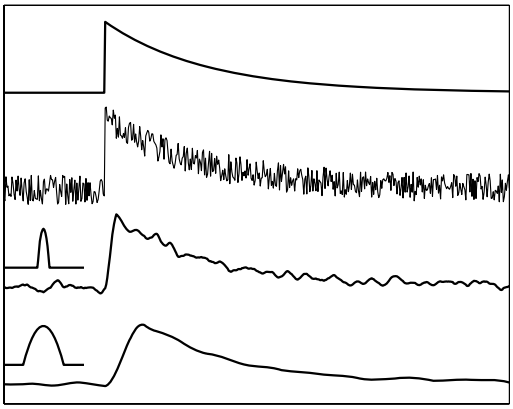


Abbildung 44: Filterung von verrauschten Messdaten mit Filterfunktionen unterschiedlicher Breite. Obere Kurve: idealer Verlauf, zweite Kurve: reales Messsignal, dritte und vierte Kurve: gefilterte Daten (Halbwertsbreite der Filterfunktionen: 6 bzw. 20 Punkte bei insgesamt 500 Datenpunkten).

Eine zweidimensionale Filterung mit `filter2` ist insbesondere bei zweidimensional ortsabhängigen Messwerten interessant. Die Glättung eines Rastertunnelmikroskopbildes veranschaulicht Abbildung 45. Links die Originalmessung, atomare Auflösung an einer Graphitprobe (HOPG[22]) mit dem bei Einfachgeräten üblichen Rauschen. Daneben das gefilterte Bild, als Filterfunktion wurde ein Rotationsparaboloid verwendet, dessen Basisdurchmesser in etwa dem atomaren Abstand entsprach.

[22] *H*ighly *O*rdered *P*yrolytic *G*raphite

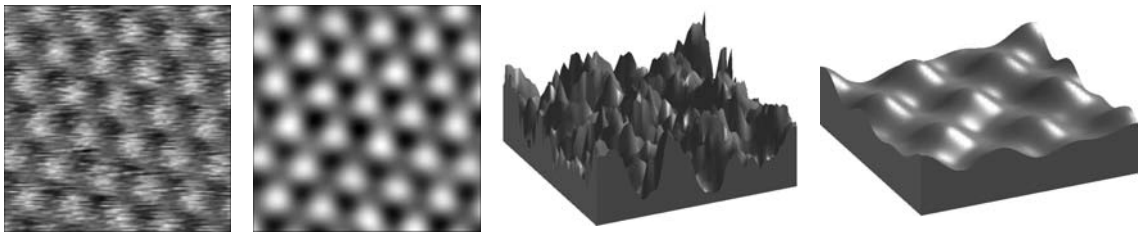


Abbildung 45: Rastertunnelmikroskopbild einer HOPG-Probe (atomare Ausflösung). Links die Originaldaten, daneben das gefilterte Bild, rechts Ausschnitte in dreidimensionaler Darstellung.

Besonders deutlich erkennt man die Wirkung des relativ drastischen Filters in den dreidimensional dargestellten Bildausschnitten rechts.

Die Konstruktion der Filterfunktion `ff` und ihre Anwendung auf die Bildmatrix `z` zeigt das folgende Fragment:

```
r0 = 10;
[xf,yf] = meshgrid(-r0:r0);
ff = r0*r0-xf.*xf-yf.*yf;
ff = ff.*(ff>0);
ff = ff/sum(sum(ff));
z = filter2(ff,z); .
```

Die Filtermatrix `ff` ist quadratisch, die Multiplikation mit `(ff>0)` in der vierten Zeile setzt die bei der Paraboloiddefinition $r_0^2 - r^2$ entstandenen negativen Matrixelemente auf null.

6.1.2 Gradientenfilter

Will man Daten nicht glätten, sondern bestimmte Detailinformationen verstärken, kann man dies mit speziell darauf zugeschnittenen Filtern erreichen. Mit einem einfachen Gradientenfilter beispielsweise lassen sich Flächen in einem Bild auf ihre Begrenzungslinien reduzieren. Das folgende Beispiel filtert ein Bild `b` mit einem Gradientenfilter für vertikale (Zeilenvektor `[-1,1]`) und einem für horizontale Gradienten (Spaltenvektor `[-1;1]`) und fasst die Ergebnisse durch eine Oder-Verknüpfung zusammen:

```
fver = [-1,1];
fhor = [-1;1];
bver = abs(filter2(fver, b));
bhor = abs(filter2(fhor, b));
bcontour = bver | bhor; .
```

Abbildung 46 zeigt die Wirkung, links das Originalbild mit seinen flächenhaften Objekten, rechts das gefilterte, auf dem nur noch die Umrisse sichtbar sind.

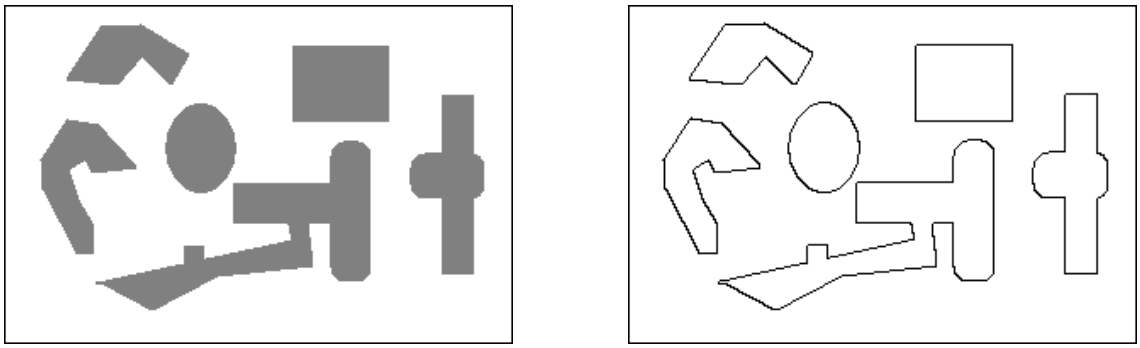


Abbildung 46: Gradientenfilter, links Original-, rechts gefiltertes Bild.

Filter dieser Art kann man dann verwenden, wenn geometrische Strukturen wie z. B. Begrenzungslinien zwischen Kristallbereichen (Domänen) automatisiert erkannt werden sollen. Man transformiert dann nach der Filterung in den Parameterraum des gesuchten geometrischen Objekts (Hough-Transformation [30]), das Maximum dort liefert die Parameter des gefundenen Objekts. Falls das Ausgangsbild nicht kontrastreich genug für das Verfahren ist, kann man sich ein Bild mit extremem Kontrast dadurch generieren, dass man alle Helligkeitswerte oberhalb eines bestimmten Grenzwerts auf 1, die darunter auf 0 setzt. MATLAB erledigt das mit

```
HighContrast = LowContrast > Threshold; .
```

Darin ist `LowContrast` die Bildmatrix mit geringem Kontrast, `Threshold` der verwendete Grenzwert und `HighContrast` das resultierende Binärbild.

6.1.3 Savitzky-Golay-Filter

In Abschnitt 6.1.1 wurde beschrieben, wie man einen geeignet gewichteten Mittelwert zur Glättung von Messdaten verwenden kann. Dabei wurde auch ein gravierender Nachteil deutlich, scharfe Strukturen in Messdaten werden verbreitert. Besser einstellen lässt sich das, wenn man statt des Mittelwerts ein Polynom geeigneter Ordnung[23] verwendet, um Daten zu glätten. Das Prinzip ist in Abbildung 47 dargestellt.

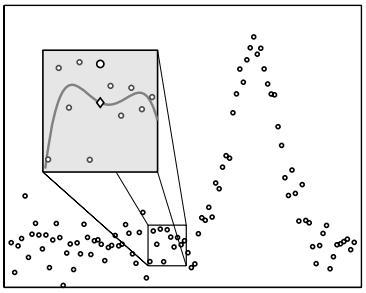


Abbildung 47: Gleitender Polynomfit: Für die Datenwerte in der Umgebung eines Punktes wird das Regressions-Polynom berechnet, der Datenpunkt (großer Kreis) wird durch den Polynomwert (Raute) ersetzt.

[23] Der einfache Mittelwert ist ein Fit-Polynom *nullter* Ordnung.

Das Verfahren scheint sehr aufwendig zu sein, für jeden Datenpunkt muss ein Regressions-Polynom berechnet werden. Man kann jedoch zeigen, dass dies nicht nötig ist und dass stattdessen eine geeignete Filterfunktion verwendet werden kann. Diese Art der Filterung wird nach den Autoren der ersten umfassenden Arbeit zu diesem Thema [31] – Abraham Savitzky und Marcel J. E. Golay – benannt.

Das Regressions-Polynom für die Umgebung des Datenpunktes i sei definiert durch

$$p_i(x) = \sum_{k=0}^{M} B_k \left(x - x_i\right)^k \,. \tag{6.3}$$

Die Polynomkoeffizienten B_k ergeben sich aus der Minimalisierung der Summe der Abstandsquadrate (*least squares fit*)

$$D_i = \sum_{j=i-n_L}^{i+n_R} \left(p_i(x_j) - y_j\right)^2 \stackrel{!}{=} \text{Min.} \tag{6.4}$$

Dabei wurde angenommen, dass das Polynom für n_L linksseitige und n_R rechtsseitige zusätzliche Punkte berechnet wird. Das Minimum wird dadurch bestimmt, dass man die partiellen Ableitungen nach allen B_m berechnet und zu Null setzt

$$\frac{\partial D_i}{\partial B_m} = 0\,, \qquad m = 0 \ldots M\,. \tag{6.5}$$

Bei Messdaten kann man als Vereinfachung fast immer annehmen, dass die Datenpunkte äquidistant sind

$$x_j - x_i = (j-i)\Delta x\,, \quad j - i = n\,, \quad n = -n_L \ldots 0 \ldots n_R\,. \tag{6.6}$$

Damit wird dann aus Gleichung 6.3

$$p_i(x_j) = \sum_{k=0}^{M} B_k n^k \Delta x^k = \sum_{k=0}^{M} b_k n^k\,, \qquad n = j - i\,. \tag{6.7}$$

Die partiellen Ableitungen (Gl. 6.5) liefern $M+1$ Gleichungen ähnlicher Struktur

$$\sum_{n=-n_L}^{n_R} n^m \left(\sum_{k=0}^{M} b_k n^k - y_{i+n} \right) = 0\,, \qquad m = 0 \ldots M\,, \tag{6.8}$$

$$\sum_{n=-n_L}^{n_R} n^m \sum_{k=0}^{M} b_k n^k = \sum_{n=-n_L}^{n_R} n^m y_{i+n}\,. \tag{6.9}$$

Das lineare Gleichungssystem 6.9 für die b_k könnte nun mit den üblichen Methoden gelöst werden. Das müsste für jeden Datenpunkt i gemacht werden.

Zur weiteren formalen Betrachtung schreibt man das lineare Gleichungssystem 6.9 in Matrixschreibweise:

$$\begin{pmatrix} (-n_L)^M & \cdots & n_R^M \\ \vdots & \ddots & \vdots \\ -n_L & \cdots & n_R \\ 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} (-n_L)^M & \cdots & -n_L & 1 \\ \vdots & \ddots & \vdots & \vdots \\ n_R^M & \cdots & n_R & 1 \end{pmatrix} \begin{pmatrix} b_M \\ \vdots \\ b_0 \end{pmatrix} = \\ = \begin{pmatrix} (-n_L)^M & \cdots & n_R^M \\ \vdots & \ddots & \vdots \\ -n_L & \cdots & n_R \\ 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_{i-n_L} \\ \vdots \\ y_{i+n_R} \end{pmatrix} \quad . \tag{6.10}$$

Man sieht, dass die ersten beiden Matrizen zueinander transponiert sind, mit

$$A = \begin{pmatrix} (-n_L)^M & \cdots & -n_L & 1 \\ \vdots & \ddots & \vdots & \vdots \\ n_R^M & \cdots & n_R & 1 \end{pmatrix} \quad , \qquad A^T = \begin{pmatrix} (-n_L)^M & \cdots & n_R^M \\ \vdots & \ddots & \vdots \\ -n_L & \cdots & n_R \\ 1 & \cdots & 1 \end{pmatrix} \tag{6.11}$$

$$\text{und} \qquad b = \begin{pmatrix} b_M \\ \vdots \\ b_0 \end{pmatrix} \qquad \text{sowie} \qquad y = \begin{pmatrix} y_{i-n_L} \\ \vdots \\ y_{i+n_R} \end{pmatrix} \tag{6.12}$$

vereinfacht sich die Schreibweise deutlich:

$$A^T A\, b = A^T y\,. \tag{6.13}$$

A ist eine Rechteckmatrix mit $M+1$ Spalten und n_L+n_R+1 Zeilen. Jede Rechteckmatrix lässt sich in eine Orthogonalmatrix Q und eine rechte Dreiecksmatrix R aufspalten (QR-Zerlegung):

$$A = Q\,R \qquad \text{und} \qquad A^T = (Q\,R)^T = R^T Q^T\,, \tag{6.14}$$

und mit

$$A^T A = R^T Q^T Q\,R = R^T I\,R = R^T R \tag{6.15}$$

erhält man schließlich

$$R\,b = Q^T y\,. \tag{6.16}$$

Der Polynomwert am Punkt $x_j = x_i(n = 0)$ ist b_0, er ergibt sich durch Multiplikation

$$b_0 = \frac{Q^T(M+1,:)}{R(M+1,M+1)}\, y\,. \tag{6.17}$$

Die Matrix A ist unabhängig von den Datenwerten y_j, damit sind dies auch die Matrizen Q und R. Die Polynomwerte ergeben sich durch eine gleitende Multiplikation (Filterung) mir einer festen Filterfunktion F_0, die Filterkoeffizienten müssen nur einmal berechnet werden

$$F_0 = \frac{Q^T(M+1,:)}{R(M+1,M+1)} = \frac{Q(:,M+1)}{R(M+1,M+1)}\,. \tag{6.18}$$

Entsprechend kann man Ableitungen berechnen, so ist die erste Ableitung am Punkt $x_j = x_i$ durch den Polynomkoeffizienten b_1 gegeben, die Filterkoeffizienten dafür sind

$$F_1 = \frac{Q(:,M) - F_0 R(M,M+1)}{R(M,M)}\,. \tag{6.19}$$

Die konkrete Berechnung der Koeffizienten war zur Zeit der Originalarbeit von Savitzky und Golay noch etwas aufwändig, der größte Teil der Veröffentlichung besteht daher aus den tabellierten Filterkoeffizienten für verschiedene Polynomordnungen und unterschiedliche Anzahl von Stützpunkten. Abbildung 48 zeigt ein Beispiel einer solchen Tabelle.

Das Arbeiten mit solchen Tabellenwerten ist relativ fehleranfällig (schon in der anscheinend computergenerierten obigen Tabelle ist mindestens ein Zahlenwert falsch). Besser ist es, die Filterkoeffizienten dann zu berechnen, wenn sie gebraucht werden. MATLAB kennt die QR-Zerlegung, dafür ist die Funktion `qr` zuständig, man muss mithin nur die Matrix A definieren [32].

Die folgende MATLAB-Funktion berechnet die benötigten Filterkoeffizienten für die Daten und die erste Ableitung, Polynomordnung M und Zahl der links- und rechtsseitigen Punkte n_L und n_R werden vorgegeben:

```
function [F0,F1,F2] = savgol(nl,nr,M)
    % coefficients for Savitzky-Golay fits
    % F0: smooth curve, F1/F2: derivatives
    % nl/nr: number of left/right points
    % M: polynomial order
A = ones(nl+nr+1,M+1);
for j = M:-1:1,
    A(:,j) = [-nl:nr]'.*A(:,j+1);
end;
[Q,R] = qr(A);
F0 = Q(:,M+1)/R(M+1,M+1);
```

CONVOLUTES	SMOOTHING			QUADRATIC	CUBIC		A20	A30			
POINTS	25	23	21	19	17	15	13	11	9	7	5
-12	-253										
-11	-138	-42									
-10	-33	-21	-171								
-09	62	-2	-76	-136							
-08	147	15	9	-51	-21						
-07	222	30	84	24	-6	-78					
-06	287	43	149	89	7	-13	-11				
-05	322	54	204	144	18	42	0	-36			
-04	387	63	249	189	27	87	9	9	-21		
-03	422	70	284	224	34	122	16	44	14	-2	
-02	447	75	309	249	39	147	21	69	39	3	-3
-01	462	78	324	264	42	162	24	84	54	6	12
00	467	79	329	269	43	167	25	89	59	7	17
01	462	78	324	264	42	162	24	84	54	6	12
02	447	75	309	249	39	147	21	69	39	3	-3
03	422	70	284	224	34	122	16	44	14	-2	
04	387	63	249	189	27	87	9	9	-21		
05	322	54	204	144	18	42	0	-36			
06	287	43	149	89	7	-13	-11				
07	222	30	84	24	-6	-78					
08	147	15	9	-51	-21						
09	62	-2	-76	-136							
10	-33	-21	-171								
11	-138	-42									
12	-253										
NORM	5175	8059	3059	2261	323	1105	143	429	231	21	35

Abbildung 48: Tabelle der Filterkoeffizienten für einen gleitenden Polynomfit zweiter Ordnung und unterschiedliche Anzahl von Stützpunkten (aus der Originalarbeit [31]).

```
F1 = (Q(:,M)-F0*R(M,M+1))/R(M,M);
F2 = (Q(:,M-1)-F0*R(M-1,M+1)-F1*R(M-1,M))/R(M-1,M-1);
```

Das folgende Skript zeigt die Anwendung auf ein Datenfeld y:

```
nl = 15;
nr = 15;
M = 6;
F0 = savgol(nl,nr,M);
sgy = filter(F0(nl+nr+1:-1:1),1,y);
sgy(1:nl) = y(1:nl);
sgy(1+nl:end-nr) = sgy(1+nl+nr:end);
sgy(end-nr+1:end) = y(end-nr+1:end);
```

Die Funktion `filter` arbeitet rückwärts gewandt, daher ist das Filter-Array `F0` gespiegelt zu verwenden. Außerdem müssen die Daten um die rechtsseitige Breite des Filters verschoben werden, um wieder zu den ursprünglichen Abszissenwerten zu passen. Zu überlegen ist auch, wie man die Randbereiche behandelt, im obigen Fragment wurden dort die Ausgangsdaten y eingesetzt.

Abbildung 49 zeigt die Anwendung von Savitzky-Golay-Filtern unterschiedlicher Ordnung auf ein verrauschtes Datenfeld im Vergleich zur einfachen Mittelung.

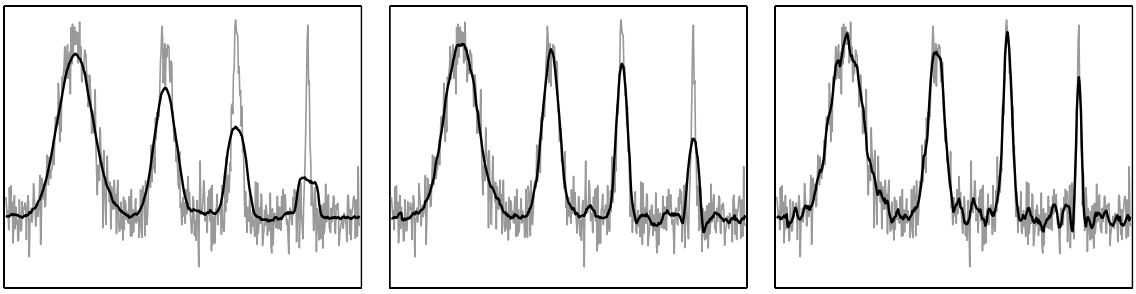


Abbildung 49: Einfache Mittelwertbildung (links) und Savitzky-Golay-Filterung zweiter (Mitte) und sechster Ordnung (rechts) bei Daten mit unterschiedlich breiten Strukturen. Grau: ursprüngliche Daten, schwarz: gefilterte Daten, Filterbreite jeweils gleich ($n_L = n_R = 15$).

6.2 Interpolation

Interpolationsverfahren werden dann verwendet, wenn man Zwischenwerte zwischen benachbarten Datenwerten benötigt. Dies ist unter anderem dann erforderlich, wenn in verwendeten Eichtabellen der erforderliche Wert nicht exakt repräsentiert ist. So enthalten Tabellen über Thermospannungen von Thermoelementen, die man zur Temperaturmessung verwendet, deren Werte nicht beliebig dicht, sondern mit irgendeiner praktikablen Schrittweite, z. B. in 10-K-Abständen, Werte dazwischen muss man geeignet interpolieren. Oft sind auch Messdaten nicht in der gewünschten Fülle gemessen worden; um dem Betrachter dennoch einen 'vollständigen' Eindruck zu vermitteln, zeichnet man interpolierte Kurven.

Zur Realisierung der Interpolation stellt MATLAB ein Reihe von Funktionen mit jeweils ähnlicher Signatur zur Verfügung.Die Anweisung

```
VI = interpN(X1,..,XN,V,XI1,..,XIN)
```

mit `N = 1,2,3,n` interpoliert die Ausgabewerte `VI` `N`-dimensional zwischen den Tabellenwerten `V`. `V` ist an den Koordinaten `X1..XN` vorgegeben, `VI` wird an den Koordinaten `XI1..XIN` berechnet.

Als weiterer Parameter kann (und sollte) die Interpolationsstrategie angegeben werden, möglich sind die folgenden Verfahren:

`'nearest'`: Zielwert wird auf den Wert des nächstliegenden Tabellenwerts gesetzt. Funktionswerte sind unstetig.

`'linear'`: Lineare Interpolation zwischen benachbarten Werten, die Funktionswerte sind stetig.

`'cubic'`: Kubische Interpolation, Funktionswerte und erste Ableitung stetig.

`'spline'`: Spline-Interpolation, Stetigkeit der Funktionswerte sowie der ersten und zweiten Ableitung.

Die Ergebnisse der verschiedenen Interpolationsverfahren hängen von der Art der zu interpolierenden Funktion bzw. der zu interpolierenden Messdaten ab. Die Abbildungen 50 und 51 veranschaulichen dies an zwei relativ extremen Beispielen: In Abbildung 50 wird eine in den Funktionswerten und Ableitungen stetige Sinusfunktion interpoliert, in Abbildung 51 eine Rechteckfunktion mit ausgeprägten Unstetigkeiten. Die Messwerte (`X,V`) werden durch die Punkte, die interpolierten Werte (`XI,VI`) durch die Kurven repräsentiert.Die interpolierte Funktionswerte wurden mit

```
VI = interp1(X,V,XI,'Strategy');
```

berechnet, für `Strategy` wurden die angegebenen Verfahren eingesetzt.

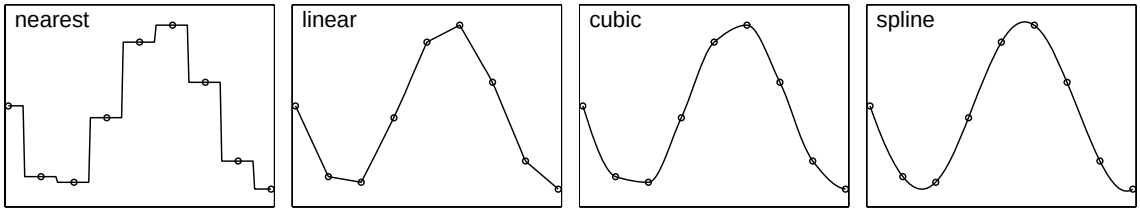


Abbildung 50: Wirkung verschiedener Interpolationsstrategien bei einer stetigen Funktion (Sinus).

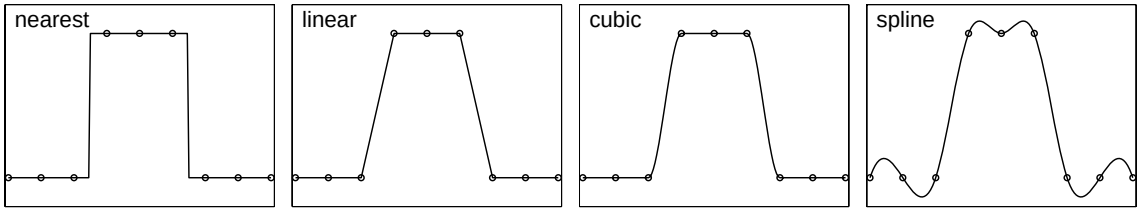


Abbildung 51: Wirkung verschiedener Interpolationsstrategien bei einer Funktion mit Unstetigkeitsstellen (Rechteck).

Der Vergleich zeigt, dass bei einer in den Werten und in den Ableitungen stetigen Funktion (Sinus) die Interpolation mit Splines sehr gut ist. Die kubische Interpolation ist ein guter Kompromiss, wenn Unklarheit über die Stetigkeit herrscht. Die `nearest`-Strategie ist dann anzuwenden, wenn Zwischenwerte nicht sinnvoll, nicht möglich oder nicht erwünscht sind. Dass sich auch die Rechenzeiten für die unterschiedlichen Verfahren beträchtlich voneinander unterscheiden, ist offensichtlich; dies kann insbesondere bei sehr großen Datenfeldern ein zusätzliches Kriterium sein.

6.3 Fouriertransformation

Messdaten, insbesondere elektrische Signale, werden meist in Abhängigkeit von der Zeit gemessen. Periodizitäten darin zeigen charakteristische Frequenzen an. Einem zeitabhängig streng periodischen Verlauf entspricht eine einzelne Frequenz, ein Punkt im Frequenzspektrum. Umgekehrt entspricht einem zeitlich einmaligen Ereignis ein Frequenzkontinuum. Die beiden Koordinaten – Zeit und Frequenz – sind zueinander komplementär.[24] Zwischen solchen komplementären Darstellungen vermittelt die Fouriertransformation. Naturgemäß spielt bei Messdaten nur die endliche diskrete Fouriertransformation eine Rolle. Zwischen den beiden komplementären Bereichen kann man Daten beliebig hin und her transformieren, jeweils die für einen bestimmten Zweck sinnvollere oder nützlichere Darstellungsform wählen.

Zur Berechnung der diskreten Fouriertransformation enthält MATLAB die Funktionen `fft`, `fft2` und `fftn`, sowie deren inverse `ifft`, `ifft2` und `ifftn` zur Rücktransformation. Die eindimensionalen Formen berechnen die Transformation in einer Dimension, auch wenn die Variable eine Matrix ist (spaltenweise bzw. in der ersten nicht singulären Dimension). Die zwei- und die mehrdimensionalen Formen rechnen entsprechend in zwei oder allen Dimensionen.

Zu den Anwendungsbereichen der Fouriertransformation gehört die Frequenzanalyse und die Datenfilterung, dazu je ein Beispiel.

6.3.1 Frequenzanalyse

Die Fouriertransformierten von streng periodischen Funktionen sind Deltafunktionen im dazu jeweils reziproken Raum. Einer zeitlich periodische Funktion entspricht mithin ein Punkt im (komplexen) Frequenzraum. Es liegt daher nahe, periodische Messdaten zu transformieren und in ihrer Frequenzdarstellung zu analysieren. Abbildung 52 zeigt als Beispiel die zu zwei verschiedenen Wähltasten – `1` und `5` – gehörenden Töne eines Telefons[25]. In der zeitlichen Auftragung ist der Unterschied nur schwer zu erkennen (linkes Bild), nach der Fouriertransformation wird die Analyse einfach (rechtes Bild).

Wie zu erwarten sind keine idealen Deltafunktionen entstanden. Die Breite der Spektrallinien entspricht der reziproken Messzeit, die hier auf 50 ms begrenzte Messzeit verursacht eine Breite von 20 Hz in der Frequenzauftragung. Ideale Deltafunktionen wären nur durch unendlich lange Messzeiten zu erreichen.

Ein zweiter bei der Fouriertransormation auftretender Effekt ist in Abbildung 52 ebenfalls erkennbar: unterschiedliche Linienformen, Restintensität neben den Linien. Hier zwar nicht sonderlich störend, da die Intensitäten der Messsignale deutlich größer sind, kann

[24] Ein ähnlich komplementäres Paar in der Physik ist beispielsweise Ort und Raumfrequenz.

[25] Bei der Tastenmatrix am Telefon sind Reihen und Spalten mit festen Frequenzen kodiert, die Reihen mit 697, 770, 852 und 941 Hz, die Spalten mit 1209, 1336 und 1477 Hz. Beim Tastendruck werden die beiden zugehörigen Frequenzen als Ton übertragen.

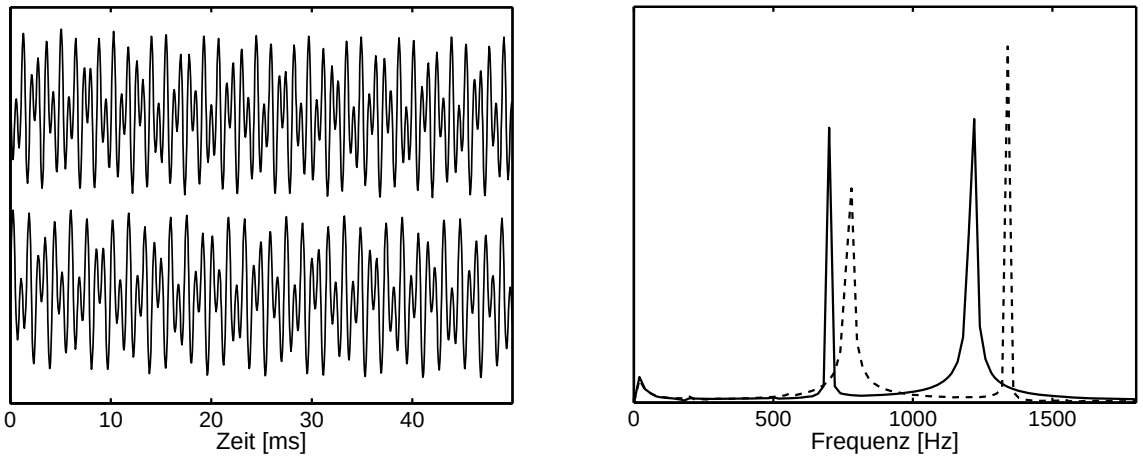


Abbildung 52: Wähltöne 1 *und* 5 *am Telefon: Links die zeitliche Auftragung der Amplituden (Oszilloskop), unten* 1*, oben* 5*, rechts die Absolutwerte der Fouriertransformierten, durchgehende Linie:* 1*, gestrichelt:* 5*.*

diese Untergrundintensität dann Probleme machen, wenn Signale analysiert werden sollen, die verschiedene Frequenzanteile mit sehr unterschiedlichen Intensitäten enthalten. Der Untergrund wird von dem stufenartigen Verlauf der Messdaten am Beginn und Ende der Messung hervorgerufen, die zusätzliche Frequenzen im Spektrum verursachen. Man vermeidet den Effekt bei der Fourieranalyse dadurch, dass man das zu analysierende Signal zuvor mit einer geeigneten Fensterfunktion multipliziert, die an den Endpunkten ganz oder nahezu verschwindet. Zur Realisierung verwendet man u. a. Dreiecks-, Trapez-, Cosinus-, Parabel- oder Gaußfunktionen. Das Verfahren hat als Nebeneffekt immer eine leichte Verbreiterung der Strukturen im Spektrum zur Folge, da durch die geringere Gewichtung der Randpunkte die effektive Messzeit verringert wird.

Die Anwendung auf das vorliegende Beispiel zeigt Abbildung 53. Es wird eine cosinusförmige Fensterfunktion verwendet, im linken Bild als graue Kurve dargestellt. Die Wirkung ist deutlich, im Spektrum ist keine Untergrundintensität mehr enthalten, die Form der Spektrallinien ist nahezu identisch geworden.

Macht man die Fensterfunktion schmaler, verkürzt damit die effektive Messzeit, so verliert man Information, die spektralen Strukturen werden breiter (Abbildung 54).

6.3.2 Datenfilterung

Außer zur Analyse von Messdaten kann man die Fouriertransformation auch dazu verwenden, Messdaten gezielt zu verändern. Ein Anwendungsbereich ist die Filterung von Daten, sofern es darum geht, Signale bestimmter Frequenzen oder ganzer Frequenzbereiche in Messdaten zu manipulieren. Man transformiert dazu den gesamten Datensatz, macht im Fourierspektrum die gewünschten Veränderungen und transformiert anschließend mit der

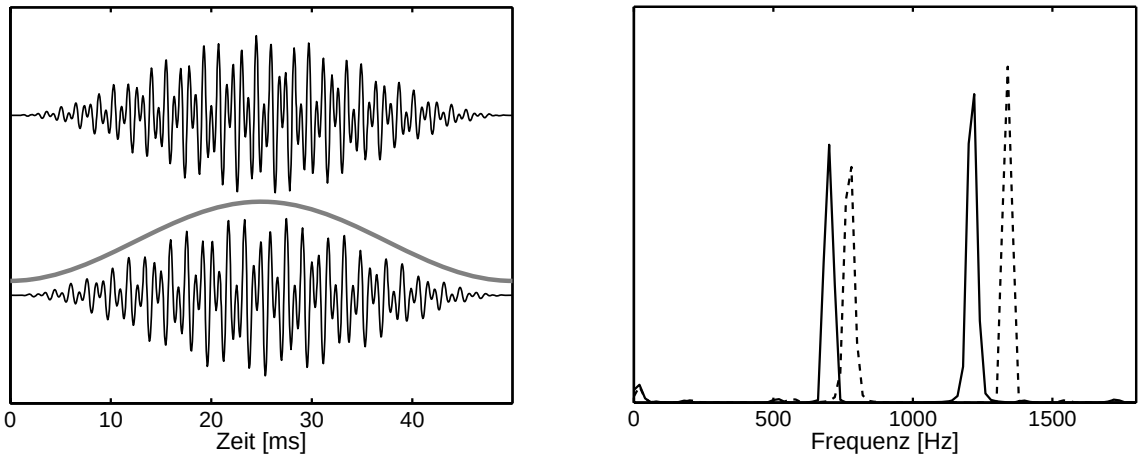


Abbildung 53: Wirkung einer Fensterfunktion bei der Fouriertransformation: Links die mit der cosinusförmigen Funktion multiplizierten Messdaten, rechts die untergrundfreien Spektren.

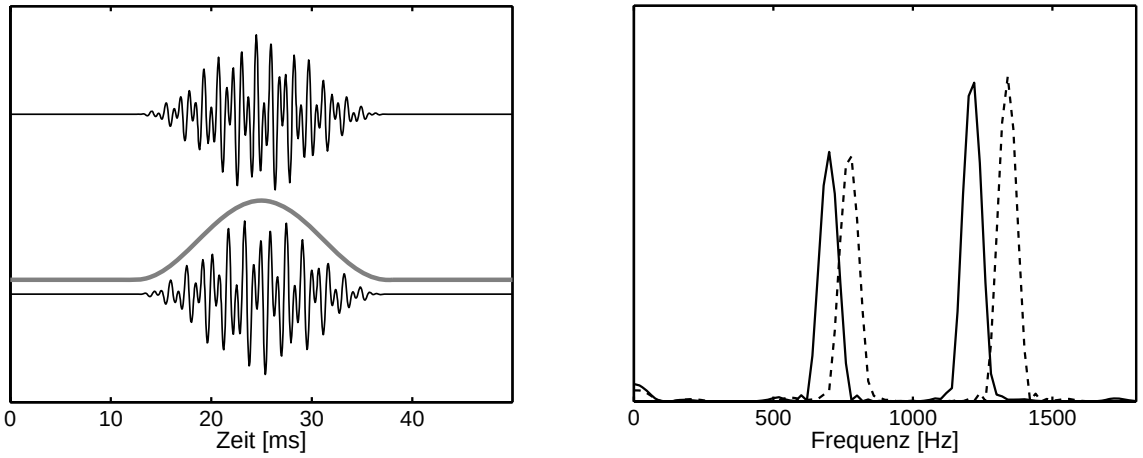


Abbildung 54: Verkürzung der effektiven Messzeit durch eine schmalere Fensterfunktion, die Spektrallinien verbreitern sich.

inversen Fouriertransformation wieder zurück.

Die Abbildung 55 zeigt als idealisiertes Anwendungsbeispiel Messdaten, denen eine intensive periodische Störung überlagert ist[26]. Eine herkömmliche Filterung (vgl. 6.1) wirkt erst dann zufriedenstellend, wenn eine relativ breite Filterfunktion verwendet wird. Dann sind aber auch die Strukturen in den Messdaten deutlich verbreitert.

Abbildung 56 zeigt die Wirkung einer Fourierfilterung. Vor der Fouriertransformation

[26] Störungen dieser Art können beispielsweise durch die Netzwechselspannung (50 oder 100 Hz) verursacht werden. Die Frequenz muss dabei nicht unbedingt der Störfrequenz entsprechen, sondern kann durch Schwebungseffekte zwischen Messtakt und Störfrequenz auch in völlig andere Bereiche verschoben sein.

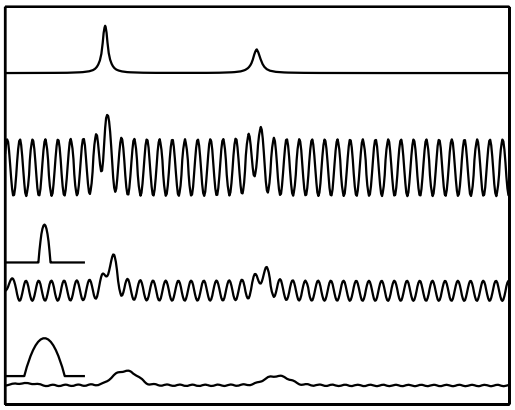


Abbildung 55: Messdaten mit einer überlagerten periodischen Störung: oben die ungestörten Daten, darunter mit Störung. Die beiden unteren Kurven zeigen das Ergebnis der in Kapitel 6.1 beschriebenen herkömmlichen Filterung der Daten mit unterschiedlich breiten Filterfunktionen (vgl. auch Abbildung 44).

werden die Messdaten mit einer Fensterfunktion multipliziert, im linken Bild als graue Kurve dargestellt. In diesem Fall wird eine Trapezfunktion verwendet, die so gewählt wird, dass sie im interessierenden Bereich der Messdaten konstant ist. Dadurch ist gewährleistet, dass die Intensitätsverhältnisse dort die Transformationen ungeändert überstehen.

Das rechte Bild (obere Kurve) zeigt das Ergebnis der Fouriertransformation (`fft`), die Störfrequenz wird durch die beiden intensiven Linien repräsentiert. Zur Unterdrückung der Störung wird im Fourierraum mit einer Funktion multipliziert, die zwischen 0 (im Bereich der Linien) und 1 (außerhalb davon) variiert (graue Kurve im rechten Bild). Auch hier wird eine trapezartige Funktion gewählt, damit keine abrupten Übergänge entstehen. Diese würden bei der Rücktransformation zu störenden Artefakten führen. Die untere Kurve im rechten Bild zeigt das Ergebnis der Multiplikation. Die Rücktransformation (inverse Funktion `ifft`) ergibt dann ein fast störungsfreies Ergebnis (untere Kurve im

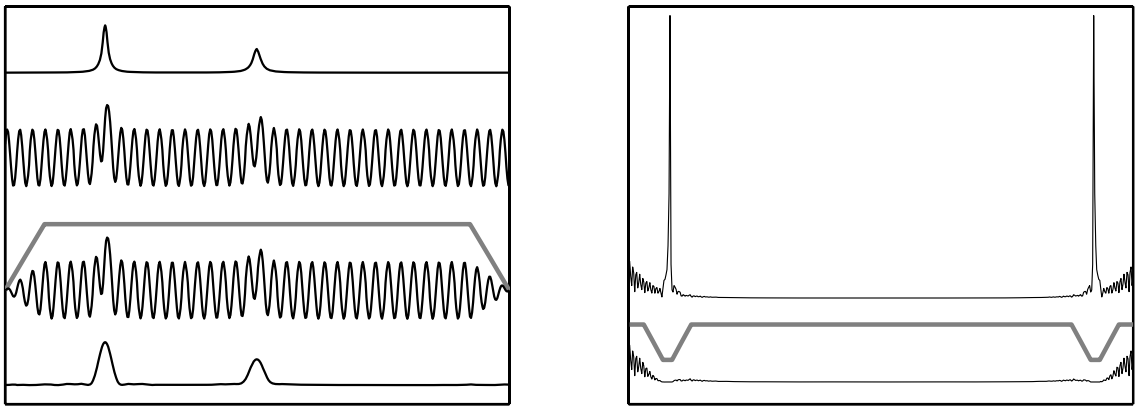


Abbildung 56: Fourierfilterung von Messdaten mit periodischer Störung. Links oben die Daten ohne und mit Störung, darunter die verwendete Fensterfunktion und die damit multiplizierten Daten. Rechts die Fouriertransformierte und die Filterfunktion, die die Störfrequenz unterdrückt. Rechts unten die damit multiplizierte Fouriertransformierte, links unten das Ergebnis der Rücktransformation.

linken Bild). Die Strukturen in den Messdaten sind etwas breiter geworden; das lässt sich nicht verhindern, da durch die Manipulation im Fourierraum auch die Bandbreite des Nutzsignals verringert wird.

6.4 Fits, Anpassung an Funktionen

Vermutet man, dass sich Messdaten durch einen mathematischen Zusammenhang beschreiben lassen, dann interessieren die Parameter, die die bekannte oder vermutete funktionale Abhängigkeit bestimmen. In diesen Parametern steckt meist die physikalische Eigenschaft, die man messen will. So kann man beispielsweise die Federkonstante einer Feder dadurch bestimmen, dass man den linearen Zusammenhang zwischen Kraft und Auslenkung misst.

Man berechnet diese Parameter dadurch, dass man die Abweichung zwischen Daten und Funktionszusammenhang möglichst klein macht (man minimalisiert die Summe der Quadrate der Differenzen – *least squares fit*). Besonders einfach ist das, wenn die vermutete Funktion ein Polynom ist oder wenn die Anpassparameter in der Funktion nur linear auftreten. Dann lässt sich das Fit-Problem exakt lösen, das Minimum ergibt sich als Lösung eines linearen Gleichungssystems. Ansonsten muss man mit geeigneten Iterationsverfahren arbeiten.

6.4.1 Polynome

Für Polynomanpassungen ist in MATLAB die Funktion `polyfit` zuständig. Der Aufruf

```
P = polyfit(x, y, N);
```

liefert im Vektor `P` die Koeffizienten des Regressionspolynoms `N`-ter Ordnung für den Datensatz `x,y`.

```
[P, S] = polyfit(x, y, N);
```

definiert eine zusätzliche Struktur `S` mit weiteren Informationen, die beispielsweise für die Fehlerabschätzung nützlich sind.

```
Yp = polyval(P, Xp);
```

berechnet die Polynomwerte an den Punkten `Xp`,

```
[Yp, dYp] = polyval(P, Xp, S);
```

zusätzlich Fehlergrenzen, innerhalb derer bei Daten mit einigermaßen zufälligem Fehler mindestens 50 % der Datenwerte liegen. Abbildung 57 zeigt als Beispiel für einen Polynomfit den einfachsten Fall, die Anpassung an eine lineare Abhängigkeit (`polyfit(x,y,1)`).

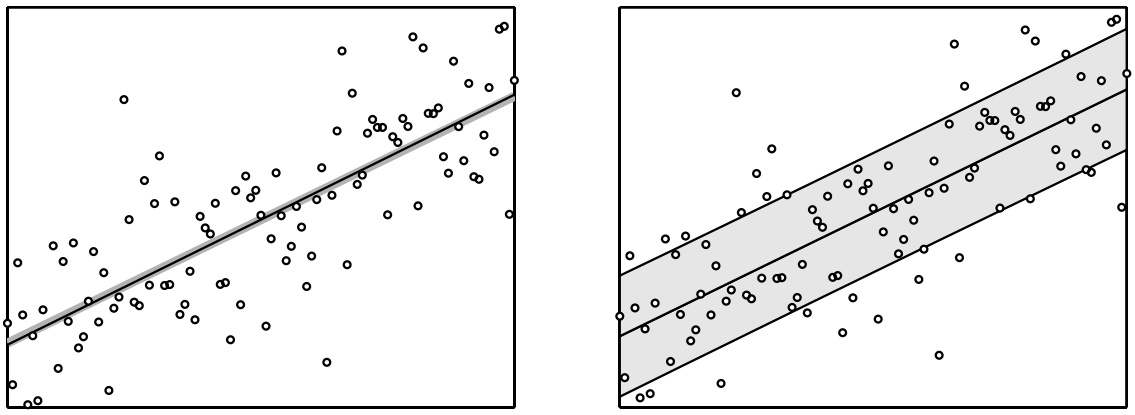


Abbildung 57: Regressionsgerade an zufallsverteilte Messdaten. Die Berechnung erfolgte mit `polyfit`*. Im linken Bild zeigt die schwarze Gerade die Ausgangsfunktion, die dann mit* `randn` *gestört wurde (Punkte), grau die Fitgerade (die leichte Abweichung deutet darauf hin, dass die Zufallszahlen nicht ganz ideal sind). Im rechten Bild ist der Fehlerbereich* `Yp-dYp...Yp+dYp` *grau hinterlegt, es wird deutlich, dass mehr als 50 % der Datenpunkte in diesem Bereich liegen.*

6.4.2 Parameterlineare Fits

Mit Matlab lassen sich die Anpassparameter auch immer dann besonders einfach bestimmen, wenn sie nur linear in der funktionalen Abhängigkeit enthalten sind, mit der die Daten gefittet werden sollen. Wenn also etwa der über x gemessene Datensatz y mit der Funktion $f(x)$ gefittet werden soll, die Anpassparameter a_i nur in linearer Form enthält:

$$y \approx f(x) = a_1 f_1(x) + a_2 f_2(x) + \cdots + a_n f_n(x) \quad . \tag{6.20}$$

Die Teilfunktionen $f_i(x)$ können darin beliebig aufwändig sein.[27]

Die Messdaten x und y sind 2 Vektoren, die je m Elemente enthalten. Da beide bekannt (gemessen) sind, kann man Gleichung 6.20 als ein lineares Gleichungssystem für die a_i auffassen. Für $m > n$, den Regelfall bei der Anpassung von Messdaten, ist das Gleichungssystem allerdings überbestimmt.

Zur Lösung linearer Gleichungssysteme hat MATLAB die Funktion `mldivide`, 'left matrix division', die verkürzt auch mit '\' (Backslash) geschrieben werden kann. Die gleiche formale Schreibweise kann auch verwendet werden, wenn das Gleichungssystem überbestimmt ist. MATLAB errechnet dann eine Näherungslösung, in der die quadratischen Abweichungen minimiert sind (least squares fit). Das ist genau das, was man benötigt.

Die Parameter a_i in Gleichung 6.20 werden berechnet durch:

$$F \quad = \quad [f_1(x), f_2(x), \ldots f_n(x)]; \tag{6.21}$$

[27] Polynome sind ein einfacher Spezialfall mit $f_1(x) = x^0$, $f_2(x) = x^1$ usw.

$$a \quad = \quad F\backslash y; \quad . \tag{6.22}$$

Das Ergebnis a ist ein Vektor mit den Parametern a_i.

Abbildung 58 zeigt ein Beispiel aus dem Bereich der nichtlinearen Optik, eine neue Substanz wurde mit Laserimpulsen unterschiedlicher Leistung beleuchtet, das erzeugte frequenzverdoppelte Licht wurde gemessen. Man erwartet einen quadratischen Zusammenhang zwischen den beiden Intensitäten $I_{2\omega}$ und I_ω:

$$I_{2\omega} = d_{\text{eff}}^2 G I_\omega^2 \quad . \tag{6.23}$$

Neben einem Geometriefaktor G enthält der Ausdruck die wirksame Nichtlinearität d_{eff}, die interessierende physikalische Größe. Die Berechnung der Fitparameter erledigt das MATLAB-Skript

```
data = load('p15_2.int');
x = data(:,1);
y = data(:,2);
F = [ones(size(x)), x.*x];
a = F\y; .
```

Zunächst werden die Messdaten gelesen, die als Tabelle in der Datei `p15_2.int` abgelegt sind, daraus werden die `x`- und `y`-Werte extrahiert. Die Funktion `F` besteht aus dem quadratischen Anteil und einer zusätzlichen Konstanten, die eine eventuell vorhandene konstante Untergrundintensität berücksichtigt. Dazu ist der mit Einsen besetzte Vektor gleicher Größe nötig. In Abbildung 58 sind Messwerte und Anpassungskurve zusammen geplottet.

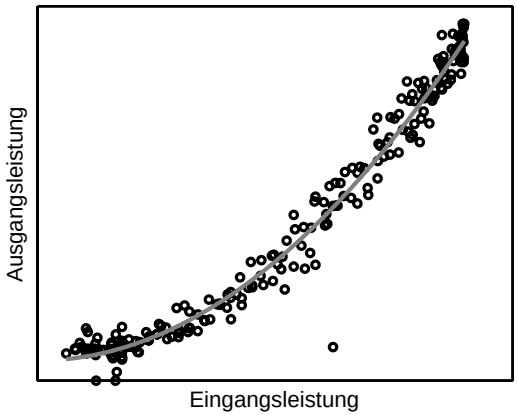


Abbildung 58: Zusammenhang zwischen Laserleistung und frequenzverdoppelter Intensität an einer nichtlinearen Probe, Messwerte (∘) und quadratische Anpassung (grau).

6.4.3 Anpassung an beliebige Funktionen

Soll eine Funktion angepasst werden, die die Fitparameter in allgemeiner Form enthält, kann das durch eine Minimalwertsuche mit der MATLAB-Funktion `fminsearch` erledigt werden. Diese Funktion wendet den Simplex-Algorithmus an, um ein Minimum zu finden.

Sie wird aufgerufen mit

```
a = fminsearch(f, a0, options, p1, p2, ...) .
```

Darin ist `a` das Ergebnis, die berechneten Parameter, `f` die zu minimierende Funktion, `a0` Startwerte[28] für `a`, `options` einstellbare Optionen wie Genauigkeit oder Maximalzahl der Iterationen, `pn` weitere Variable, die an `f` übergeben werden. Die Funktion `f` hat die Form

```
d = f(a, p1, p2, ...) .
```

Bei der Bestimmung von Fitkurven berechnet man in `f` zum Beispiel die Summe der quadrierten Differenzen zwischen Messwerten und zugehörigen Funktionswerten der Fitfunktion.

Zur Veranschaulichung wieder ein einfaches Beispiel aus der Optik. Die Abhängigkeit des Brechungsindex von der Wellenlänge kann mit guter Genauigkeit durch eine Summe von Resonanztermen ausgedrückt werden:

$$n(\lambda)^2 = 1 + \sum_i \frac{A_i}{\lambda_i^{-2} - \lambda^{-2}} \quad . \tag{6.24}$$

Für diese Terme sind jeweils Absorptionsübergänge (Elektronen, Phononen etc.) bei den Wellenlängen λ_i verantwortlich. Im sichtbaren Spektralbereich genügt es oft, nur *einen* derartigen – im Ultravioletten liegenden – Term zu berücksichtigen.

Im vorliegenden Beispiel wurden die Brechungsindizes von Benzophenon, einem optisch anisotropen Material, für die 3 Hauptpolarisationsrichtungen bei verschiedenen Wellenlängen gemessen. Die Wellenlängen waren durch die verwendeten Spektrallampen vorgegeben. Brechungsindizes in den Bereichen dazwischen könnte man durch geeignete Interpolation erhalten, sehr viel besser ist es jedoch, den durch physikalische Modellvorstellungen begründeten Funktionsverlauf anzupassen, der durch ganz wenige Parameter (in diesem Fall jeweils ein Resonanzterm mit zwei Parametern für jede der drei Polarisationsrichtungen) beschrieben werden kann.

Das MATLAB-Skript für die Anpassungsrechnung:

```
function para = coeff(fname)
start = [ 1e14, 2e-7 ];
options = optimset('TolX', 1e-10, 'TolFun', 1e-10, ...
                   'MaxIter', 1e4, 'MaxFunEvals', 1e4);
data = load(fname);
para = fminsearch('devsum', start, options, data(:,1), data(:,2)); .
```

Der Minimalisierungsalgorithmus wird mit Startwerten und Optionen aufgerufen, die Optionen werden zuvor mit `optimset` festgelegt (Toleranzen und Maximalzahl der Iterationen). An die zu minimierende Funktion `devsum` werden die Messdaten aus der Datei `fname` weitergereicht, Wellenlängen und zugehörige Brechungsindizes.

[28] Beim verwendeten Simplex-Verfahren ist es sehr wichtig, sinnvolle Startwerte vorzugeben, da der Algorithmus auch nach Auffinden eines *lokalen* Minimums abbricht.

Die Funktion `devsum` berechnet die Summe der quadrierten Abstände der Messwerte von der jeweils aktuellen Fitkurve:

```
function f = devsum(para, lambda, nexp)
f = sum((nexp - nfit(para, lambda)).^2); .
```

Die Fitkurve selbst wird in einer weiteren Funktion berechnet, um sie auch für andere Zwecke (Graphik) verwenden zu können:

```
function n = nfit(para, lambda)
n = sqrt(1+para(1)./((para(2))^(-2)-lambda.^(-2))); .
```

Die errechneten Fitkurven sind zusammen mit den Messwerten in Abbildung 59 dargestellt. Deutlich wird hierin auch ein weiterer Vorteil von (wenigparametrigen) Fitkurven gegenüber einer Interpolation: Kleine statistische Messfehler werden weitgehend ausgemittelt.

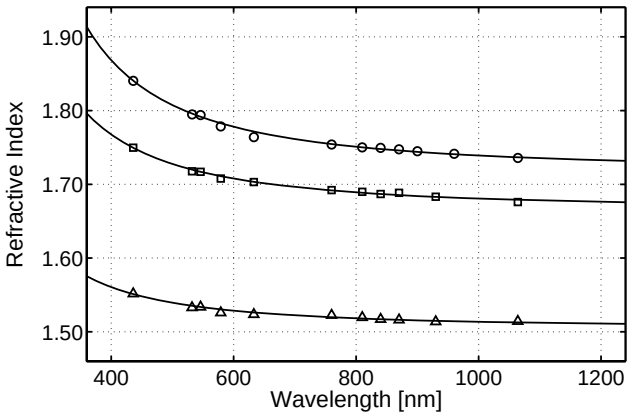


Abbildung 59: Brechungsindizes von Benzophenon, oben n_a, in der Mitte n_b, unten n_c. Die Punkte repräsentieren die Messwerte, Linien die Fitkurven.

6.5 Graphische Darstellung

Obwohl MATLAB in erster Linie für Anwendungen in der Numerik konzipiert ist, bietet es auch eine Fülle von Visualisierungsfunktionen, die zur Erstellung von Graphiken eingesetzt werden können. Darunter sind 2D-Funktionen wie `plot` für lineare 2D-Plots, `loglog` für doppelt logarithmische oder `semilogx` und `semilogy` für halblogarithmische Darstellung und 3D-Funktionen wie `plot3` für dreidimensionale Linienplots oder `surf` für Flächendarstellungen.

Alle Graphik-Objekte sind in MATLAB in einer hierarchischen Objekt-Struktur angeordnet, die einzelnen Objekte sind über ihre *Handles*[29] zugänglich. Alle Objekteigenschaften können während der Erstellung oder nachträglich mit Hilfe der *Handles* fast beliebig eingestellt werden. So wurde beispielsweise die Kurve und die Punkte für n_a in Abbildung 59 konfiguriert mit:

[29] 'Handle Graphics' ist ein eingetragenes Warenzeichen von *The MathWorks Inc.*

```
hfita = plot(L, na, 'k-');
set(hfita, 'LineWidth', 1.5);
hold on;
data = load('n_a.dat');
hna = plot(1e9*data(:,1), data(:,2), 'ko');
set(hna, 'LineWidth', 1.5);
set(hna, 'MarkerSize', 7)
```

Darin sind die ersten beiden Parameter in den `plot`-Anweisungen jeweils die Daten, `'k-'` ordnet eine durchgezogene schwarze (blac*k*) Linie an, `'ko'` schwarze kreisförmige Marker. Mit `set` werden anschließend weitere Eigenschaften festgelegt.

Eine Einführung in die Graphik-Möglichkeiten von MATLAB gibt das Skriptum zu Vorlesung 'Graphik-Workshop' [29], eine Fülle von Informationen findet man außerdem im MATLAB-Hilfesystem und in der Online-Hilfe unter den Einstiegspunkten `help graphics`, `help graph2d` und `help graph3d`.

7 Schnittstellen und Programmierung

Die Kopplung zwischen Experiment und Steuerungs- bzw. Auswerterechner wird im allgemeinen durch Schnittstellen (*Interfaces*) vermittelt – Standardschnittstellen, die an praktisch jedem Rechner gleichartig vorhanden sind, oder aber speziellen Schnittstellenkarten, die für bestimmte Aufgaben gebaut sind. Eigenschaften und Progammierung der Rechnerschnittstellen sind Gegenstand dieses Kapitels, das Schwergewicht liegt dabei auf den Standardschnittstellen. Die Programmbeispiele werden in MATLAB, C/C++ oder Java erstellt.

> Programmiersprachen sind keine Weltanschauungen sondern Werkzeuge;
> es gibt geeignete und ungeeignete und man darf sie ungestraft wechseln.

7.1 Die serielle Schnittstelle

Serielle Verbindungen sind die vom Leitungsaufwand her einfachsten genormten 2-Punkt-Verbindungen in der DV-Technik. Ursprünglich sind sie für langsame, zeichenorientierte Datenübertragung zwischen Fernschreibern oder zwischen Rechner und Terminal konzipiert. Am PC ist meist zumindest eine serielle Schnittstelle vorhanden, an die Peripheriegeräte wie Maus, Modem, Plotter, (serielle) Drucker, Datenerfassungsgeräte oder auch ein anderer Rechner angeschlossen werden können.

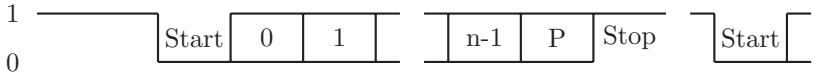
7.1.1 Grundlagen und Schnittstellennorm

Damit Geräte verschiedener Hersteller ohne weitere Anpassungsarbeit miteinander kommunizieren können, müssen alle relevanten Parameter herstellerübergreifend festgelegt sein. Für serielle Verbindungen sind mehrere Normen definiert, die sich in der Hardwareauslegung unterscheiden. Je nach Norm sind unterschiedliche Maximalentfernungen und Maximalübertragungsgeschwindigkeiten möglich. In der zur Zeit gebräuchlichsten Norm (USA: RS 232 C, D: DIN 66020, 66022, Europa: CCITT V24)[30], die auch bei der PC-Schnittstelle verwendet wird, sind unter anderem die folgenden Parameter vereinbart:

- Der High-Pegel einer Leitung liegt zwischen +3 und +15 Volt, der Low-Pegel zwischen -3 und -15 Volt.
- Daten werden in negativer Logik (1 = Low-Pegel), Steuersignale in positiver Logik (True, aktiv = High-Pegel) übertragen.
- Der Ruhezustand der Datenleitung ist logisch 1. Bei der asynchronen Übertragung werden Zeichen einzeln übertragen, die einzelnen Bits eines Zeichens in Folge. Jedes

[30] Die Nummern der Normblätter werden häufig auch zur Bezeichnung der Schnittstelle verwendet: RS-232-Interface, V24-Schnittstelle.

Bit hat die gleiche zeitliche Länge. Die Übertragung wird durch ein auf logisch 0 gesetztes Bit, das Startbit, eingeleitet, dann folgen die Datenbits in steigender Wertigkeit, danach eventuell ein Paritätsbit P, am Ende mindestens 1–2 Bits logisch 1 (Stopbits, = Ruhezustand der Leitung).



Die Zeichenlänge (Wortlänge) n liegt zwischen 5 (Fernschreiber) und 8 Bit (8 Bit ASCII[31]).

- Die Übertragungsgeschwindigkeit, das ist die Zahl der übertragenen Bits pro Sekunde[32], muss zwischen Sender und Empfänger vereinbart werden, damit sich der Empfänger nach dem Startbit auf die ankommende Bitfolge synchronisieren kann. Üblich sind $75 \cdot 2^N$ bit/sec mit $N = 0 \cdots 9$, d. h. 75, 150, 300, ... 9600, 19200, 38400 bit/sec, in Sonderfällen werden aber auch andere Übertragungsgeschwindigkeiten eingestellt[33].

- Neben den beiden Datenleitungen für Senden (TxD) und Empfangen (RxD) sind einige Steuerleitungen definiert, die anzeigen, ob das betreffende Gerät eingeschaltet ist (DSR, DTR), empfangsbereit ist (CTS, RTS), die Leitung in Ordnung ist (DCD) oder ein Anruf angekommen ist (RI). Ein Teil dieser Leitungen ist für den Modem-Betrieb und damit die Datenfernübertragung gedacht, im lokalen Betrieb können sie zur Datenflusssteuerung (Quittungsbetrieb) benutzt werden.

- Als Steckverbindungen wurden bis vor einigen Jahren 25-polige D-Stecker und -Buchsen benutzt, bei neueren PCs fast ausschließlich 9-polige, auf Spezialkarten teilweise auch kleinere. Die Steckerbelegung ist davon abhängig, ob es sich um ein Datenendgerät (DTE, Data Terminal Equipment) oder um ein Modem-artiges Gerät (DCE, Date Communication Equipment) handelt. PCs fungieren immer als DTE-Geräte, die Steckerbelegung dafür ist in Tabelle 3 angegeben. Bei DCE-Geräten ist die Signalrichtung (Ein/Aus) komplementär.

  Verbindungskabel zwischen einem DTE- und einem DCE-Gerät verbinden jeweils gleiche Pin-Nummern, bei Kabeln zwischen 2 DTE-Geräten (z. B. zwischen PC und Plotter) müssen die sich entsprechenden Pins (RxD↔TxD, CTS↔RTS) jeweils 'gekreuzt' verbunden werden(Abbildung 60). Oft genügen zum Anschluss die Datenleitungen und Masse, teilweise werden die Steuerleitungen zur Datenflusskontrolle benutzt.

[31] ASCII = American Standard Code for Information Interchange

[32] Für bit/sec ist die Einheit Baud gebräuchlich, die Übertragungsgeschwindigkeit wird oft als Baud-Rate bezeichnet.

[33] Moderne Bausteine können mit Übertragungsraten bis zu 1 MBaud betrieben werden.

Pin 9-pol.	Pin 25-p.	Ein/ Aus	Signal
1	8	E	DCD, Data Carrier Detected, Leitung in Ordnung
2	3	E	RxD, Receive Data, Dateneingang
3	2	A	TxD, Transmit Data, Datenausgang
4	20	A	DTR, Data Terminal Ready, Terminal betriebsbereit
5	7		GND, Signal Ground, Masseanschluss
6	6	E	DSR, Data Set Ready, externes Gerät betriebsbereit
7	4	A	RTS, Request To Send, Sendeanforderung zum externen Gerät
8	5	E	CTS, Clear To Send, Sendeanforderung vom externen Gerät
9	22	E	RI, Ring Indicator, Wählsignal vom MODEM

Tabelle 3: Steckerbelegung der seriellen Schnittstelle im PC

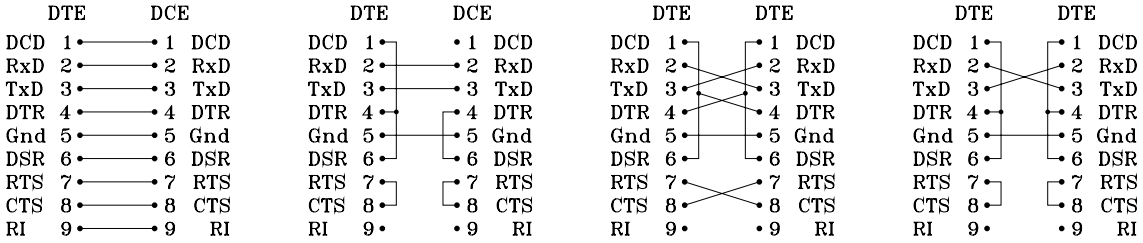


Abbildung 60: Die wichtigsten Kabeltypen für serielle Verbindungen vom PC zu einem Peripheriegerät, Pinbelegungen für 9-polige Stecker bzw. Buchsen. Von links: Volle DTE-DCE-Verbindung (Rechner ⇄ Modem), DTE-DCE ohne Steuerleitungen, volle DTE-DTE-Verbindung (Nullmodemkabel zwischen zwei PCs), DTE-DTE ohne Steuerleitungen.

7.1.2 Quittungsbetrieb

Werden die Daten über eine serielle Leitung potentiell schneller übertragen, als sie von einem der Teilnehmer verarbeitet werden können (Drucker, Plotter), muss eine Möglichkeit vorgesehen werden, den Datenfluss zu steuern (Quittungsbetrieb, Handshake). Zwei Arten der Datenflusssteuerung werden überwiegend verwendet:

- Beim 'Hardware-Handshake' wird durch den Status der Steuerleitungen mitgeteilt, ob ein Gerät empfangsbereit ist. Die meisten Geräte benutzen zu diesem Zweck die CTS- bzw. RTS-Leitung (CTS-RTS-Handshake). Das Sendergerät muss vor der Ausgabe jedes einzelnen Zeichens den Steuerleitungsstatus überprüfen und gegebenenfalls den 'Aktiv'-Zustand der Handshake-Steuerleitung abwarten.

- Beim 'Software-Handshake' (auch XON-XOFF-Handshake) sind zwei Zeichen vereinbart, die vom Empfängergerät zum Stop und zur Wiederaufnahme der Übertragung ans Sendergerät geschickt werden. Gestoppt wird mit 'XOFF' (meist CTRL-S), wiedergestartet mit 'XON' (meist CTRL-Q). Das Sendergerät sollte insbesondere auf XOFF prompt reagieren, d. h. die Datenausgabe sofort anhalten.

Bei der Übertragung von größeren Datenmengen zwischen Rechnern ist es oft sinnvoll, im Blockbetrieb zu arbeiten. Dabei bildet das Sendergerät nach einem vereinbarten Algorithmus einen Block aus einer bestimmten Anzahl von Zeichen, der zusammen mit einer Prüfsumme verschickt wird. Der Empfänger prüft den Block anhand der Prüfsumme auf Richtigkeit und quittiert positiv oder negativ. Bei negativer Quittung wird der Block nochmals geschickt. Auf diese Weise lässt sich eine hohe Datensicherheit erreichen. Die meisten klassischen Datenübertragungsprogramme arbeiten mit solchen 'Protokollen' (Kermit, Crosstalk, X-Modem, ...).

7.1.3 Andere Übertragungsnormen

Die RS 232 C-Schnittstellen-Norm ist relativ alt und ursprünglich für niedrige Übertragungsraten konzipiert. Laut Spezifikation ist sie für Leitungslängen bis zu 15 m und für Übertragungsraten bis zu 20 kbit/s ausgelegt. Inzwischen sind deutlich höhere Übertragungsraten möglich und gebräuchlich (MODEMs, Rechner-Rechner-Kopplung). Neben der RS 232 C existieren zwei wesentlich leistungsfähigere Normen, die allerdings im PC-Bereich wenig verwendet werden.

Die Norm RS 423 A definiert eine unsymmetrische Schnittstelle, bei der die Datenübertragung über ein (mit dem Wellenwiderstand) abgeschlossenes Koaxialkabel erfolgt. Die maximale Übertragungsrate ist 300 kbit/s, die maximale Leitungslänge 600 m.

Die Norm RS 422 A benutzt symmetrische Leitungstreiber und -empfänger und abgeschlossene, verdrillte Zweidrahtleitungen. Die maximale Leitungslänge ist 1200 m, die maximale Übertragungsrate 2 Mbit/s (bei dann allerdings verringerter Leitungslänge von max. 60 m).

Noch höhere Übertragungsraten sind durch Lichtleiterverbindungen, größere Übertragungsstrecken durch Modulationsverfahren (Telefon-Modem) oder über die einschlägigen digitalen Postdienste (Datex-P, ISDN, DSL) realisierbar.

Für alle Normen sind Treiberbausteine verfügbar, die eine Umsetzung des vom Schnittstellenbaustein generierten TTL-Signals auf die Norm-Signale vornehmen.

7.1.4 Programmierung unter Windows

Bis vor einigen Jahren konnte die serielle Schnittstelle des PC mit akzeptabler Geschwindigkeit nur durch direkte Programmierung der Schnittstellenbausteine betrieben werden.

Zum einen lag das an der beschränkten Funktionalität der Bausteine, zum andern an den wenig leistungsfähigen Betriebssystemfunktionen. Beides hat sich deutlich geändert: in der Entwicklung von den 'klassischen' ICs 8250 und 16450 zum 16550 bzw. Multifunktionsbausteinen mit integriertem 16550 wurde der Zeichenpuffer im Baustein vergrößert, und moderne Betriebssysteme wie Windows 32 bieten alle zum Betrieb notwendigen Funktionen. Direkte Portprogrammierung [33] ist daher nicht mehr sinnvoll.

Vom Betriebssystem wird die serielle Schnittstelle logisch wie eine Datei gehandhabt, die gelesen und geschrieben werden kann. Folglich muss zunächst ein Datei-Objekt angelegt werden, um die Eigenschaften der Schnittstelle einstellen zu können. Unter Win32 gibt es dafür die Funktion `CreateFile`:

```
HANDLE hCOM = CreateFile("COM1:", 0, 0, NULL, OPEN_EXISTING, 0, NULL); .
```

Danach können mit den Methoden `ReadFile` und `WriteFile` Datenblöcke von der Schnittstelle gelesen bzw. zur Schnittstelle geschickt werden, mit `CloseHandle` wird die Schnittstelle wieder freigegeben.

Die Schnittstellenparameter sind in einem im Betriebssystem vorgehaltenen umfangreichen Bit-Feld, dem *Device Control Block*, definiert:

```
typedef struct _DCB { // dcb
  DWORD DCBlength;           // sizeof(DCB)
  DWORD BaudRate;            // current baud rate
  DWORD fBinary: 1;          // binary mode, no EOF check
  DWORD fParity: 1;          // enable parity checking
  DWORD fOutxCtsFlow:1;      // CTS output flow control
  DWORD fOutxDsrFlow:1;      // DSR output flow control
  DWORD fDtrControl:2;       // DTR flow control type
  DWORD fDsrSensitivity:1;   // DSR sensitivity
  DWORD fTXContinueOnXoff:1; // XOFF continues Tx
  DWORD fOutX: 1;            // XON/XOFF out flow control
  DWORD fInX: 1;             // XON/XOFF in flow control
  DWORD fErrorChar: 1;       // enable error replacement
  DWORD fNull: 1;            // enable null stripping
  DWORD fRtsControl:2;       // RTS flow control
  DWORD fAbortOnError:1;     // abort reads/writes on error
  DWORD fDummy2:17;          // reserved
  WORD wReserved;            // not currently used
  WORD XonLim;               // transmit XON threshold
  WORD XoffLim;              // transmit XOFF threshold
  BYTE ByteSize;             // number of bits/byte, 4-8
  BYTE Parity;               // 0-4=no,odd,even,mark,space
  BYTE StopBits;             // 0,1,2 = 1, 1.5, 2
  char XonChar;              // Tx and Rx XON character
  char XoffChar;             // Tx and Rx XOFF character
```

```
  char ErrorChar;            // error replacement character
  char EofChar;              // end of input character
  char EvtChar;              // received event character
  WORD wReserved1;           // reserved; do not use
} DCB;
```

Die Bedeutung der einzelnen Komponenten sowie vordefinierte Konstanten für die Zuweisung von Werten sind z. B. in der Online-Hilfe zu Visual C++ ausführlich beschrieben (suchen unter 'DCB').

Um Einstellungen zu ändern, wird der *Device Control Block* kopiert, geändert und anschließend wieder zurückkopiert, etwa so:

```
DCB dcb;
GetCommState ( hCOM, &dcb );
dcb.BaudRate = CBR_38400;              // set to 38400 Baud
...
SetCommState ( hCOM, &dcb );
```

Durch eine entsprechende Kontrollstruktur wird das Verhalten der Schnittstelle bei Zeitüberschreitung (*Timeout*) definiert (Online-Hilfe unter 'COMMTIMEOUTS'):

```
typedef struct _COMMTIMEOUTS {   // ctmo
  DWORD ReadIntervalTimeout;
  DWORD ReadTotalTimeoutMultiplier;
  DWORD ReadTotalTimeoutConstant;
  DWORD WriteTotalTimeoutMultiplier;
  DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

Änderungen ähnlich wie oben durch:

```
COMMTIMEOUTS ctmo;
GetCommTimeouts ( hCOM, &ctmo );
ctmo.ReadIntervalTimeout = 100;      // allow 100 msec between
...                                  //   arriving characters
SetCommTimeouts ( hCOM, &ctmo );
```

Zusätzlich zu dieser umfassenden Art der Schnittstellenprogrammierung gibt es vereinfachte Funktionen, um die Ausgangsleitungen der Schnittstelle statisch zu setzen:

```
BOOL SetCommBreak ( HANDLE hFile );   // handle of comm. device
BOOL ClearCommBreak ( HANDLE hFile ); // handle of comm. device
```

setzen die Datenleitung auf logisch 0 bzw. logisch 1.

```
BOOL EscapeCommFunction ( HANDLE hFile,   // handle of comm. device
                          DWORD dwFunc ); // function to perform
```

mit `dwFunc = SETRTS, CLRRTS, SETDTR, CLRDTR, ...` setzt die Quittungsleitungen in definierte Zustände.

Mit diesen Funktionen können die Ausgangsleitungen der Schnittstelle relativ einfach zur Steuerung von primitiven Geräten genutzt werden, beispielsweise wie in folgendem Programmfragment zum Schalten eines Halbleiterrelais:

```
void CAnyDlg::SwitchOn()
  { EscapeCommFunction ( hCOM, SETRTS ); };

void CAnyDlg::SwitchOff()
  { EscapeCommFunction ( hCOM, CLRRTS ); };
```

Neben den bisher beschriebenen Funktionen gibt es eine Anzahl weiterer, die dazu dienen, die serielle Schnittstelle ereignisgesteuert zu verwenden (`SetCommMask`, `WaitCommEvent`, ...). Da sie im Umfeld der Messdatenerfassung relativ selten benötigt werden, sollen sie hier nicht näher diskutiert werden (sie sind in der Online-Hilfe ausführlich dokumentiert).

7.1.5 C++ und Microsoft Foundation Classes

Objektorientiert in C++ kann man die Schnittstelle – z. B. unter Visual-C++ – programmieren, wenn man die Klassenbibliothek der *Microsoft Foundation Classes* verwendet. Man konstruiert ein `CFile`-Objekt, etwa

```
CFile  SerialLine ( "COM1:", CFile::modeReadWrite ); ,
```

dessen Methoden `Read` und `Write` dann zur Ein- und Ausgabe von Datenblöcken aufgerufen werden.

Die zum Zugriff auf den *Device Control Block* und die *Timeout*-Struktur benötigte *Handle* ist in der Klasse `CFile` als Variable definiert:

```
HANDLE hCOM = SerialLine.m_hFile .
```

7.1.6 C-Programmierung mit *Stream-IO*-Funktionen

Die Windows-Funktionen `ReadFile` und `WriteFile` sind für binäre Datenblöcke gedacht, ebenso die entsprechenden Methoden der Klasse `CFile`. Die Formatierung erfolgt getrennt davon, etwa mit `sprintf` bzw. `sscanf`. Statt der Windows-Funktionen kann man aber auch die von C gewohnten *Stream-IO*-Funktionen verwenden, die die Formatierung mit erledigen (`fprintf`, ...). Die Schnittstelle wird dabei wie eine Datei gehandhabt. Im folgenden Fragment werden Textanweisungen formatiert ausgegeben und Daten binär gelesen (Tektronix-Oszilloskop):

```
FILE * com = fopen("COM1:", "r+");
setvbuf(com, NULL, _IONBF, 0);
fprintf(com, "DAT:SOU CH%d\r\n", channel);
```

```
...
fprintf(com, "CURV?\r\n");
fflush(com);
fread(buffer, sizeof(char), 10, com);
...
fclose(com); .
```

Die Funktion `setvbuf` weist den Compiler an, keinen Datenpuffer bereitzustellen, die Daten also direkt zur Schnittstelle weiterzuleiten. Da dies nicht von allen Compilern strikt befolgt wird, ist zusätzlich ein `fflush` zwischen Schreiben und Lesen zu empfehlen, um den eventuell doch vorhandenen Puffer zu leeren.

Sollen Schnittstellenparameter über den *Device Control Block* eingestellt werden, muss dazu eine kurze Programmsequenz

```
HANDLE hCOM = CreateFile ( "COM1:",  ... );
DCB dcb;
GetCommState ( hCOM, &dcb );
...
CloseHandle ( hCOM );
```

– wie in 7.1.4 beschrieben – vorgeschaltet werden.

7.1.7 Linux-Spezifisches

Unter Linux kann die serielle Schnittstelle ebenfalls mit *Stream-IO*-Funktionen betrieben werden, statt `"COM1:"`, ... sind als Gerätenamen die entsprechenden Linux-*Devices* `"/dev/ttyS0"`, ... einzusetzen.

Desweiteren ändert sich die Programmierung der Schnittstellenparameter, dem *Device Control Block* entspricht unter Linux die etwas andere Struktur `termios`; hier ein Anwendungsbeispiel:

```
FILE * com = fopen("/dev/ttyS0", "r+");
int fn = fileno(com);
struct termios options;
tcgetattr(fn, &options);             // get current options
cfsetispeed(&options, B9600);        // inputrate
cfsetospeed(&options, B9600);        // outputrate
cfmakeraw(&options);                 // raw input
options.c_cflag |= (CLOCAL | CREAD); // local usage, receive
options.c_cflag |= CRTSCTS;          // enable HardwareFlowControl
options.c_cc[VTIME] = 10;            // timeout (*0.1s)
options.c_cc[VMIN] = 0;              // minimum received chars
tcsetattr(fn, TCSANOW, &options);    // activate the changes now.
```

7.1.8 Programmierung in MATLAB

Seit dem Release 12 (September 2000) enthält Matlab eine Funktionsbibliothek für die seriellen Schnittstellen. Mit der Funktion `serial` wird zunächst ein Objekt erstellt, dessen Parameter Matlab-üblich mit `set` und `get` eingestellt oder erfragt werden können[34]. Daneben ist auch ein objektartiger Zugriff in Punktschreibweise möglich.

Inzwischen (ab Release 13 – Juni 2002) ist in Matlab auch ein *Property Inspector* integriert, mit dem Objekt-Eigenschaften interaktiv inspiziert und verändert werden können. Zumindest in der Testphase ist das ein Hilfsmittel, das einem viel Arbeit ersparen kann. Der *Property Inspector* wird mit

```
inspect(ser);
```

aufgerufen, nachdem das serielle Objekt beispielsweise durch

```
ser = serial('COM1');
```

erstellt wurde.

Auf das Schnittstellenobjekt werden dann transparent die C-ähnlichen Matlab-I/O-Funktionen angewendet. Das folgende Beispiel – eine Funktion `osc`, die Daten aus einen Digitalspeicheroszilloskop (Tektronix 210) in Matlab einliest – verdeutlicht die Verwendung:

```
function y = osc(channel)
ser = serial('COM1');
ser.FlowControl = 'hardware';
ser.InputBufferSize = 3000;
ser.Terminator = 'CR/LF';
fopen(ser);
fprintf(ser, 'SEL:CH%d ON\n', channel);
fprintf(ser, 'DAT:SOU CH%d\n', channel);
fprintf(ser, 'DAT:ENC RPB\n');
fprintf(ser, 'CURV?\n');
a = 0;
while char(a)~='#',
    a = fread(ser, 1, 'uchar');
end;
b = fread(ser, 1, 'uchar');
n = fread(ser, str2num(char(b)), 'uchar');
y = fread(ser, str2num(char(n')), 'uint8');
fclose(ser);
delete(ser);
clear ser;
```

[34] Die komplette Liste der möglichen Parameter erhält man mit `set(handle)`, die Liste der aktuellen Werte mit `get(handle)`, `handle` ist die von `serial` zurückgelieferte *Object Handle*.

Das Schnittstellenobjekt wird mit `serial` erstellt, anschließend werden einige Parameter festgelegt – Datenflusskontrolle, Puffergröße, Zeilenendezeichen. Dann werden nach `fopen` mit `fprintf` einige Anweisungen an das Oszilloskop gegeben (Kanalauswahl, Datenformat), die letzte (`CURV?`) fordert die aktuellen Daten an. Gelesen wird solange, bis ein #-Zeichen kommt, danach werden die Daten relevant. Zunächst eine Ziffer (`b`), die angibt, aus wieviel Ziffern die folgende Zahl – `n` – besteht[35]. Diese wiederum informiert über die Größe des Datenblocks `y`. Die Folge von `fread`-Funktionen entspricht dieser Datenstruktur. Die letzten 3 Zeilen räumen auf.

7.2 Direkte Port-Ein/Ausgabe unter Windows 32

Während die serielle Schnittstelle komfortabel mit Betriebssystemfunktionen zu bedienen ist, müssen andere Schnittstellen meist relativ maschinennah programmiert werden. Dazu ist es sehr hilfreich, wenn man aus einem Datenerfassungsprogramm mit C-Funktionen[36] direkt auf die zugehörigen Ein/Ausgabe-Ports zugreifen kann. In einem Multi-Tasking-Betriebssystem, das außerdem ein gewisses Sicherheitsniveau verspricht, ist dies nicht mehr selbstverständlich. Dazu zunächst ein Zitat aus einem neueren Werk über Visual C++ [34]:

> ***Zugriff auf den physikalischen Speicher und die E/A-Schnittstellen***
>
> *Programmierer der 16-Bit-Version von Windows waren damit vertraut, direkt auf den physikalischen Speicher oder die Eingabe-/Ausgabe-Schnittstellen zuzugreifen. . . .*
>
> *Für 32-Bit-Betriebssysteme ist diese Vorgehensweise nicht mehr möglich. Win32 ist ein plattformunabhängiges Betriebssystem. Alle plattformabhängigen Elemente sind vollständig inkompatibel mit dem Betriebssystem. Dazu zählen alle Arten des Zugriffs auf die physikalische Hardware, wie z. B. Schnittstellen, physikalische Speicheradressen usw.*
>
> *Wie aber schreiben Sie Anwendungen, die direkt mit der Hardware kommunizieren können? Dazu benötigen Sie eines der verschiedenen DDKs (Device Driver Kits). Über ein DDK kann eine Treiber-Bibliothek erzeugt werden, die den gesamten Low-Level-Zugriff auf das Gerät enthält und Ihre High-Level-Applikation von allen Plattformabhängigkeiten befreit.*

Ende des Zitats.

Der Autor hat insofern recht, als Gerätetreiber die einzig sichere Art sind, auf Schnittstellen zuzugreifen, da man nur auf diese Weise Zugriffskonflikte verhindern kann. Andererseits lässt Windows 98 (und wohl auch ME) den Low-Level-Zugriff auf die Schnittstellen zu, und Visual C++ bietet nach wie vor die notwendigen Funktionen zur Programmierung.

[35] Mit `fread` wird `n` als Spaltenvektor gelesen, daher das transponierte `n'` bei der Umwandlung in eine Zahl in der folgenden Zeile.

[36] In Microsoft C/C++ sind das die Funktionen `_inp, _inpw, _inpd` für die Eingabe und `_outp, _outpw, _outpd` für die Ausgabe eines Bytes, Worts oder Doppelworts.

Verwenden sollte man diese Möglichkeit jedoch nur, wenn man Zugriffskonflikte auf andere Weise unterbindet. So sollte man darauf achten, dass Messprogramme, die solche Zugriffe verwenden, nicht mehrfach auf einem Rechner gestartet werden.

7.2.1 Portzugriff unter Windows 2000

Schwieriger wird die Sache unter Windows 2000: ein Programm mit Low-Level-Zugriffen, das unter 98 oder ME problemlos arbeitet, wird von 2000 gestoppt, weil es gegen die Systemsicherheit verstößt. Ohne Gerätetreiber ist unter 2000 kein Zugriff auf die Schnittstellen möglich. Frei verfügbare *generische* Gerätetreiber, die den Zugriff auf Ports (und auch auf den Speicher) pauschal ermöglichen, können einem jedoch die Arbeit abnehmen, eigene Treiber zu schreiben.

Ein Beispiel für einen solchen Gerätetreiber ist `UNIIO`, der von der Firma *BBD SOFT* als freie Software zur Verfügung gestellt wird [35]. *BBD SOFT* ist dabei, *'Unified IO - C++ interface for industrial IO cards'* zu entwickeln. Ziel ist es, plattformunabhängige C++-Klassen für industrielle Peripheriekarten zu entwickeln. Plattform- und Compiler-abhängig müssen dabei nur noch kurze Routinen für den direkten Hardwarezugriff implementiert werden.

Das umfangreiche Softwarepaket ist derzeit mit Treibern für Windows 2000 und Linux sowie darauf aufsetzenden Klassen für verschiedene Peripheriekarten frei verfügbar [35]. Als Compiler unter Windows wird vom Hersteller allerdings bisher nur Visual C++ unterstützt.

Für eigene Anwendungen reicht es aus, einen kleinen Teil des Gesamtpakets zu verwenden. Im Wesentlichen ist das die Klasse `IOPort` (Dateien `ioport.hpp` und `ioport.cpp`), die die direkten Port-Ein/Ausgabe-Funktionen anonymisiert. Dazu wird noch die Klasse `OSInterface` (Dateien `osiface.hpp` und `osiface.cpp`) benötigt, dort wird in der Windows-Implementierung der Gerätetreiber `uniio.sys` aktiviert, der den Zugriff auf Ports und Speicher freigibt. Der Treiber muss dafür zuvor vom Administrator im System installiert sein, das kann das im Paket enthaltene Installationsprogramm `loaddrv.exe` erledigen.

Unter Visual C++ sind die Klassen dann direkt verwendbar, für Dev-C++ sind Anpassungen nötig. Dev-C++ hält sich strikt an den ANSI-Standard, enthält mithin keine Port-Ein/Ausgabe-Funktionen. Die Erweiterung lässt sich am einfachsten durch eine Anleihe bei Linux bewerkstelligen. Die Datei `asm_io.h` enthält alles Nötige. Leider heißen die Funktionen anders als bei Microsoft – und die Parameterreihenfolge ist unterschiedlich[37]. Dem

```
_outpw ( PortNumber, Value );
```

[37] Dies hängt damit zusammen, dass die ersten Assembler von Intel und Motorola eine unterschiedliche Reihenfolge von Quell- und Zieloperand verwendeten. Microsoft ist in der Tradition von Intel, Gnu in der von Motorola.

bei Microsoft entspricht ein

```
outw ( Value, PortNumber );
```

bei Gnu. Man muss also zusätzlich die Funktionen in der Klasse `IOPort` anpassen. Weiterhin ist als Compiler-Option `-D_ANONYMOUS_STRUCT` einzufügen, da ein verwendeter Speicheradresstyp eine anonyme Struktur enthält.

7.2.2 Test: PC-Lautsprecher

Ohne zusätzliche Hardware lassen sich die Port-Ein/Ausgabe-Funktionen am einfachsten mit dem in jedem PC eingebauten Lautsprecher testen. Für dessen Ansteuerung ist der *Timer2* des im PC – zumindest als Funktion – eingebauten Timer-Bausteins 8254 mit der Basisadresse 0x40 zuständig (Genaueres dazu in [33] Seite 18 ff.).

Die Frequenz des Rechtecksignals für den Lautsprecher wird mit

```
void SetFreq (int freq)
{
  unsigned short Divisor = 1193180 / freq;
  IOPort ModePort (0x43);
  ModePort.writeChar (0xb6);
  IOPort FreqPort (0x42);
  FreqPort.writeChar (Divisor);
  FreqPort.writeChar (Divisor >> 8);
}
```

eingestellt. Der *Timer2* wird damit auf Rechteckmodus programmiert, die Taktfrequenz des Timers von 1193180 Hz wird durch den eingestellten Divisor auf die gewünschte Frequenz geteilt.

Basierend auf dem Ton A mit 440 Hz werden die Frequenzen für weitere Töne in *temperierter* Stimmung berechnet – darin ist das Frequenzverhältnis zweier aufeinanderfolgender Töne $\sqrt[12]{2}$ (Halbtonschritt):

```
void PlayNote (int note, int duration)
{
  SetFreq ((int)(440 * pow(2, note / 12.0)));
  IOPort EnPort (0x61);
  EnPort.writeChar (EnPort.readChar() | 0x03);  // start speaker
  Sleep (duration);
  EnPort.writeChar (EnPort.readChar() & ~0x03); // stop speaker
}
```

Die beiden niederwertigsten Bits von Port 0x61 aktivieren den Lautsprecher.

Das Hauptprogramm übernimmt Tonnummer und Tondauer (in ms) aus der Kommandozeile, ohne Parameter wird eine Sekunde ein C gespielt.

```
int main (int argc, char * argv[])
{
  int note = 3, duration = 1000;
  if (argc>2) {
    note = atoi(argv[1]);
    duration = atoi(argv[2]);
    }
  PlayNote(note, duration);
  return 0;
}
```

Notwendiger Programmvorspann:

```
#include <cmath>
#include <stdlib.h>
#include "osiface.hpp"
#include "ioport.hpp"
using namespace Uniio;
```

Benötigt man statt eines eigenständigen Programms eine MEX-Funktion, so ist nur obige `main`-Funktion zu ersetzen durch

```
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  if (nrhs==0) return;
  double * y = mxGetPr(prhs[0]);
  int M = mxGetM(prhs[0]);
  for (int i=0; i<M; i++)
    PlayNote((int)*(y+i), (int)*(y+M+i));
}
```

Ein Musikstück kann dann als Matrix mit 2 Spalten (Tonnummer und -dauer) und M Zeilen vorgegeben werden.

7.2.3 Bit-Operationen

Beim Zugriff auf Ein/Ausgabe-Ports ist es oft notwendig, einzelne Bits der Ein- oder Ausgabedaten auf einen bestimmten Wert zu setzen oder zu überprüfen. C stellt dafür einige Bit-Operationen zur Verfügung, unter C++ kann man stattdessen die Klasse `bitset` verwenden.

C-Stil. Die Vorgehensweise ist in [36] beschrieben (Kapitel 6.9: Bit-Felder):

> *Dies geschieht üblicherweise, indem man „Bit-Masken“ definiert, die den relevanten Bit-Positionen entsprechen, also*

```
#define KEYWORD   01
#define EXTERNAL  02
#define STATIC    04
```

oder

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Die Zahlen müssen dabei Potenzen von 2 sein. Zugriff auf die Bits wird dann ein Problem, das mit den Operatoren für Bit-Manipulation gelöst werden muss, die in Kapitel 2 beschrieben wurden.

Bestimmte Redewendungen sind sehr häufig. Die Zuweisung

```
flags |= EXTERNAL | STATIC;
```

setzt die EXTERNAL- und STATIC-Bits in flags auf Eins,

```
flags &= ~(EXTERNAL | STATIC);
```

löscht genau diese Bits, und die Bedingung

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

ist genau dann erfüllt, wenn beide Bits gelöscht sind.

Dem ist allenfalls hinzuzufügen, dass man sich zur einfacheren Handhabung der Zweierpotenzen ein Makro etwa der Art

```
#define BIT(b) (1<<b)
```

definieren kann, das Fehler vermeidet und gleichzeitig dokumentiert:

```
enum { KEYWORD = BIT(0), EXTERNAL = BIT(1), STATIC = BIT(2) }; .
```

C++: bitset. Im ANSI[38]-Standard für C++ vom Dezember 1996 [37] ist die generische Klasse (*template class*) `bitset` definiert (Kapitel 23.3.5), die alle notwendigen Methoden zur Bit-Manipulation enthält (vgl. auch Online-Hilfe zu Visual C++). Ein 16-Bit-Objekt würde deklariert mit

```
std::bitset<16> flags; ,
```

ein einzelnes Bit (Bit b, $0 \leq b \leq 15$) mit

`flags.set(b);` oder `flags.reset(b);` bzw. `flags.flip(b);`

auf Eins oder Null gesetzt bzw. negiert. Neben anderen Operatoren ist insbesondere auch der Zugriffsoperator `[]` definiert, der mit

`flags[b] = 1;` oder `flags[b] = 0;` bzw. `flags[b] = !flags[b];`

die gleiche Wirkung erzielt. Das Testen eines Bits kann ebenfalls auf (mindestens) zwei Weisen erfolgen;

`if (flags.at(b) == 0) ...` oder `if (flags[b] == 0) ...`

Zum Arbeiten mit Bitsets sollte man sich passende anonymisierte Ein/Ausgabe-Funktionen bereitstellen, etwa für Visual C++:

```
void OUTB (USHORT Port, std::bitset<16> Data)
```

[38] American National Standards Institute.

```
    { _outp (Port, (int) Data.to_ulong()); };
```

oder für Gnu C++:

```
void OUTB (USHORT Port, std::bitset<16> Data)
    { outb ((int) Data.to_ulong(), Port); }; .
```

Bei Verwendung der UNIIO-Bibliothek würde man zusätzlich die Klasse `IOPort` passend erweitern:

```
IOPort & IOPort::writeChar ( const std::bitset<16> Data )
    { OUTB (thePortNumber, Data); } .
```

7.3 Zeit und Windows

Windows ist kein 'Echtzeit'-Betriebssystem, daher zur exakten zeitlichen Steuerung eines Ablaufs im Prinzip nicht geeignet. Wenn man aber kleinere Ungenauigkeiten in Kauf nehmen kann, sind die Windows-Funktionen zur Zeitmessung und Zeittaktsteuerung recht vielseitig verwendbar.

7.3.1 Systemzeit

Die Rechneruhr wird unter Windows im Abstand von 10 ms inkrementiert, mit dieser Auflösung und Genauigkeit arbeiten auch alle Zeitfunktionen, die darauf zugreifen (auch wenn die Zeitangaben in Millisekunden sind). Unter anderem sind dies:

`Sleep` – setzt die Programmausführung für die als Parameter (in Millisekunden) angegebene Zeit aus.

`clock` – berechnet die seit dem Programmstart vergangenen Zeittakte. Die formale Auflösung ist durch die Konstante `CLOCKS_PER_SEC` gegeben, die reale Auflösung ist die der Rechneruhr. Die Zeit in Sekunden ergibt sich durch Division von `clock` durch `CLOCKS_PER_SEC`.

`GetTickCount` – liefert die seit dem Start von Windows vergangene Zeit (in Millisekunden) als 32-Bit-Wert (`DWORD`)[39].

7.3.2 Zeitmessung in MATLAB

Dazu äquivalente Funktionen sind auch in MATLAB verfügbar, ebenfalls mit einer Auflösung von 10 ms:

[39] Bei Rechnersystemen, die länger in Betrieb sind, muss gegebenenfalls berücksichtigt werden, dass der Millisekunden-Zähler etwa alle 50 Tage überläuft und wieder bei Null anfängt.

`pause` – wartet für die (diesmal in Sekunden anzugebende) Zeit.

`clock` – liefert Datum und Zeit als Vektor mit 6 Elementen `[Jahr Monat Tag Stunden Minuten Sekunden]`.

`etime` – berechnet die Differenz aus zwei `clock`-Werten.

`tic, toc` – starten und stoppen eine Zeitmessung, `toc` liefert die Zeitdifferenz.

`cputime` – liefert die Zeit (in Sekunden) seit dem Start von MATLAB.

7.3.3 Performance-Counter

Zur genaueren Zeitmessung kann unter Windows ein spezieller Zähler verwendet werden, der ein hochfrequentes Taktsignal zählt. Dieser Zähler wird mit der Funktion

```
QueryPerformanceCounter(LARGE_INTEGER * N)
```

ausgelesen, die Frequenz des zugehörigen Taktsignals kann man mit

```
QueryPerformanceFrequency(LARGE_INTEGER * F)
```

erfragen. Ist kein entsprechender Zähler in der Rechner-Hardware vorgesehen, wird als Frequenz 0 gemeldet. Typischerweise liegen die verwendeten Frequenzen zwischen 1 MHz und der Prozessortaktfrequenz. Mithin sind Zeitmessungen mit Mikrosekundengenauigkeit oder besser möglich.

Die Messung einer Zeitdifferenz `delta` mit dem Performance Counter veranschaulicht das nachstehende Programmfragment:

```
LARGE_INTEGER f, n1, n2;
QueryPerformanceFrequency(&f);
QueryPerformanceCounter(&n1);
...
QueryPerformanceCounter(&n2);
double delta = (double) (n2.QuadPart-n1.QuadPart)/f.QuadPart; .
```

Will man nicht die Differenz der Ablesezeitpunkte messen, sondern nur die dazwischen liegende Zeit, beispielsweise um die Ausführungszeit von Programmcode zu bestimmen, dann sollte man noch mit einer Leermessung korrigieren. Eine Beispielanwendung für den Performance-Counter ist in Abschnitt 7.4.3 beschrieben.

7.3.4 Timer

Prozesse unter Windows können sich Taktgeber (Timer) einrichten, die in festem zeitlichen Abstand entweder eine dafür bereitgestellte Funktion aufrufen oder an das zugehörige Fenster die `WM_TIMER`-Nachricht verschicken. Sie sind immer an ein Fenster-Objekt (Window) gebunden, können daher nicht in reinen Konsolprogrammen eingerichtet werden.

Timer werden mit `SetTimer()` eingerichtet, mit `KillTimer()` wieder entfernt. Die Zeitauflösung und damit auch die minimal einstellbare Taktrate beträgt derzeit 10 msec. Nachstehendes Programmfragment verdeutlicht die Verwendung:

```
static UINT MyTimer = 1;
const UINT Elapse = 100;                              // timeout 100 msec
...
void CALLBACK TiProc (HWND hWnd, UINT msg, UINT timer, DWORD systime)
{ Do what you want to do every 100 msec }
...
MyTimer = SetTimer (hWnd, MyTimer, Elapse, TiProc); // install timer
...
KillTimer (hWnd, MyTimer);                            // deinstall timer
```

7.3.5 Timer in MATLAB

Seit der Version 6.5 (Release 13) stehen auch in Matlab Timer-Objekte zur Verfügung, die mit dem Aufruf

```
t = timer('TimerFcn', @mycallback, 'Period', 10.0);
```

eingerichtet werden. Eigenschaften des Timer-Objekts werden direkt bei der Einrichtung festgelegt oder später – Matlab-üblich – entweder mit

```
set(t,'Eigenschaft1',Wert1,'Eigenschaft2',Wert2,...);
```

oder mit

```
t.Eigenschaft = Wert; .
```

`get(t)` und `set(t)` zeigen die aktuellen und die möglichen Werte für die Objekteigenschaften an.

Die Zeitauflösung entspricht der des umgebenden Betriebssystems, bei Windows derzeit 0.01 s.

Die Verwendung von *Callback*-Funktionen ist sehr flexibel, neben der üblichen *TimerFcn* können Funktionen zugewiesen werden, die beim Starten (*StartFcn*), Stoppen (*StopFcn*) oder Fehlern (*ErrorFcn*) des Timers aufgerufen werden. Nachdem ein Timer-Objekt mit allen Eigenschaften eingerichtet ist, wird es mit `start(t)` gestartet, mit `stop(t)` wieder angehalten. Weitere Informationen dazu in der Matlab-Hilfe.

7.3.6 Multimedia-Timer

Für Anwendungen, bei denen die Zeitauflösung der 'normalen' System-Timer nicht ausreicht, stellt Windows einen weiteren Timer-Typ bereit, den Multimedia-Timer mit einer Auflösung von 1 ms. Neben der höheren Auflösung hat dieser Timertyp den Vorteil, nicht an Fenster-Objekte gebunden zu sein, er kann somit auch in Konsolprogrammen, fensterlosen DLLs (MEX) u. ä. verwendet werden. Gestartet wird mit

```
MMRESULT timeSetEvent( UINT uDelay, UINT uResolution,
                       LPTIMECALLBACK lpTimeProc,
                       DWORD dwUser, UINT fuEvent );
```

`uDelay` ist die Verzögerung (Taktrate) in Millisekunden, `uResolution` die Auflösung (Genauigkeit), `lpTimeProc` die Callback-Funktion, die vom Timer aufgerufen werden soll, `dwUser` Daten, die an die aufgerufene Funktion übergeben werden, `fuEvent` legt fest, ob der Timer einmalig (`TIME_ONESHOT`) oder periodisch arbeiten soll (`TIME_PERIODIC`).

Der Rückgabewert identifiziert den Timer und wird zum Stoppen des Timers mit

```
MMRESULT timeKillEvent( UINT uTimerID );
```

benötigt (`uTimerID`).

Die Callback-Funktion, die vom Multimedia-Timer aufgerufen wird, muss dem folgenden Prototyp entsprechend implementiert werden

```
void CALLBACK TimeProc( UINT uID, UINT uMsg,
                        DWORD dwUser, DWORD dw1, DWORD dw2 );
```

darin identifiziert `uID` den Timer und `dwUser` enthält die beim Timer-Start übergebenen Daten, die übrigen Parameter sind reserviert oder undefiniert.

Mit

```
MMRESULT timeGetDevCaps( LPTIMECAPS ptc, UINT cbtc );
```

können die Fähigkeiten (minimale und maximale Zeit) der Multimedia-Timers erfragt werden.

Ein Beispielprogramm, in dem Funktionen des Multimedia-Timers verwendet werden, ist in Abschnitt 7.4.2 beschrieben.

7.4 Parallele Schnittstellen

werden benutzt, um Peripheriegeräte über mehrere Leitungen gleichzeitig anzusprechen und um Daten schnell byte- oder wortweise einzulesen oder auszugeben. Ein Standardgerät, das im allgemeinen auf diese Weise angeschlossen ist, ist der Drucker. Die Druckerschnittstelle arbeitet mit 8 Datenleitungen (ein Byte wird jeweils komplett zum Drucker übertragen), einer Leitung, die anzeigt, dass die Daten gültig sind und einer Leitung, die

meldet, ob der Drucker beschäftigt ist. Weitere Leitungen sind für zusätzliche Statusmeldungen zuständig. In den letzten Jahren wurde die Funktionalität der Druckerschnittstelle beträchtlich erweitert, so dass bei modernen Rechnern eine sehr schnelle bidirektionale Datenübertragung zu Peripheriegeräten wie externen Disk-Laufwerken, Scannern o. ä. möglich ist.

Bei einfachen konfigurierbaren Parallel-Schnittstellen-Karten (die also nicht ausschließlich für einen Verwendungszweck vorgesehen sind wie die Druckerschnittstelle) wurde überwiegend der Baustein 8255 verwendet, für festgelegte Spezialanwendungen sind oft Sonderanfertigungen notwendig (schnelle Kopplung von externen Geräten an den PC).

7.4.1 Die Druckerschnittstelle alter Art

wird, da ursprünglich nur für einen speziellen Zweck – die Ansteuerung des Druckers – vorgesehen, über festverdrahtete, nicht weiter konfigurierbare Ausgabe- und Eingabe-Register betrieben. Einen schematischen Überblick über die Schaltung und die Zuordnung der Register-Bits zur 25-poligen Buchse am PC gibt Abbildung 61.

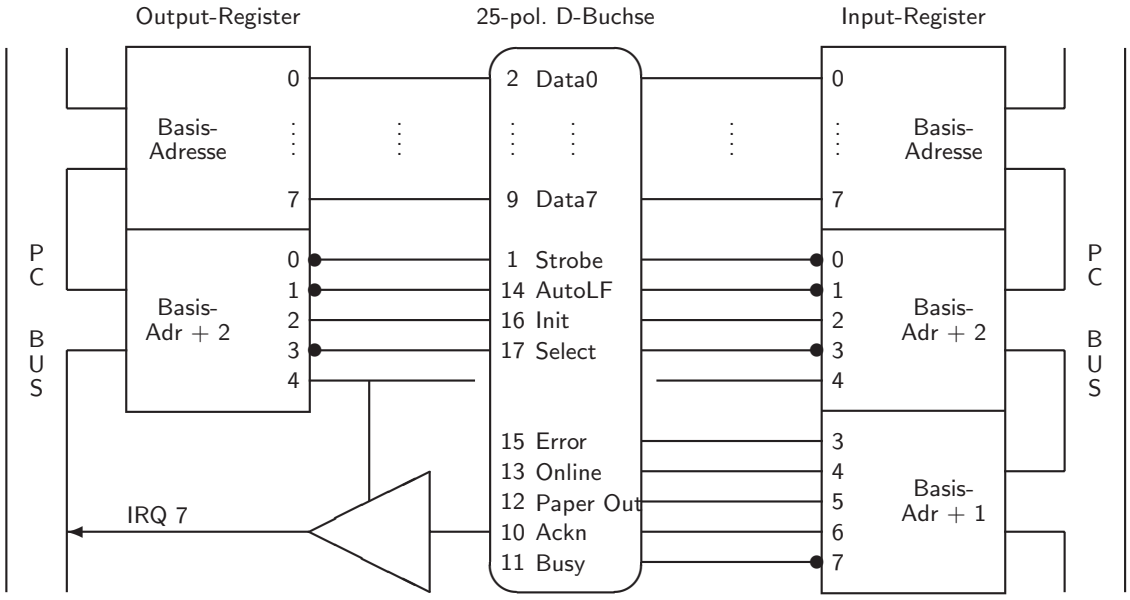


Abbildung 61: Schaltung des Druckeranschlusses im PC, Pins 18–25 der 25-poligen D-Buchse sind mit Masse verbunden. • : Invertierung des Signals.

Über die Basisadresse der Druckerschnittstelle[40] werden die 8 Datenleitungen zum Drucker

[40] Üblicherweise ist dies 0x378 für die erste Druckerschnittstelle; bei modernen Boards können über das ROM-BIOS auch andere Standardadressen eingestellt werden. Die Adressen der am PC vorhandenen Druckerschnittstellen werden beim Systemstart im RAM ab 0x408 vermerkt.

angesprochen, ihr Zustand kann über diese Adresse festgelegt und gelesen werden. Das Ausgaberegister mit Basisadresse+2 bedient mit den Bits 0–3 die zum Drucker gehenden Kommandoleitungen. Bit 4 dieses Registers legt fest, ob die vom Drucker kommende Acknowledge-Leitung auf die IRQ7-Leitung des PC zugreifen und damit einen Interrupt auslösen kann. Auch der Zustand dieser Leitungen kann — über Basisadresse+2 — eingelesen werden. Die vom Drucker kommenden Statusleitungen sind mit den Bits 3–7 eines Eingaberegister verbunden, das unter Basisadresse+1 gelesen werden kann.

Durch die Druckerschnittstelle sind somit in jedem PC 12 Ausgabe- und 5 Eingabeleitungen vorhanden, eine Eingabeleitung kann als Interruptleitung benutzt werden. Die Signale sind TTL-kompatibel, d. h. logisch 1 ist nominell +5 V, logisch 0 nominell 0 V. Zumindest die Datenleitungen und die Strobeleitung lassen relativ niederohmigen Betrieb zu ($> 150\ \Omega$). Verwendet man paarweise verdrillte Kabel oder Flachbandkabel zum Anschluss, so ist durch die Anordnung der Masseleitungen ein sehr störsicherer Betrieb bis zu Frequenzen von mehr als 100 kHz möglich. Mithin ist die Druckerschnittstelle auch gut für Steuerungszwecke – beispielsweise zur Ansteuerung von Schrittmotoren – zu gebrauchen.

7.4.2 Druckerport und Multimedia-Timer zur Schrittmotoransteuerung

Im folgenden Beispiel wird die Ansteuerung eines Schrittmotors über das Druckerport implementiert. Es werden zwei Bits des Ports benutzt, um Schrittmotortakt (Bit 0) und Laufrichtung (Bit 1) vorzugeben. Einfache Schrittmotornetzgeräte erzeugen daraus die für den Motorbetrieb notwendige Abfolge von Stromimpulsen.

Zum Betrieb des Druckerports wird die Klasse `lptPort` definiert:

```
class lptPort {
private:
  unsigned short BaseAddress;
  enum {DATA = 0, STATUS = 1, CONTROL = 2};
  int Bit(int n) { return 1<<n; };
public:
  lptPort(const char * name = "lpt1");
  void writeData(const int data);
  int readData();
  void setBit(const int n);
  void resetBit(const int n);
  int readStatus();
  void writeControl(const int data);
}; .
```

Der Konstruktor initialisiert die Basisadresse des verwendeten Druckerports (`prn`, `lpt1`, ...), sie ist im Speicherbereich 0x408 ff. vermerkt. Dazu werden die Speicherzugriffsfunktionen aus dem UNIIO-Paket verwendet (Dateien `physmem.hpp` und `physmem.cpp`):

```
using namespace Uniio;
lptPort::lptPort(const char * name) : BaseAddress(0)
{
  PhysicalMemory aMemory (0x00400, 0x10, false);
  if ((stricmp(name, "lpt")==0) || (stricmp(name, "prn")==0)
      || (stricmp(name, "lpt1")==0))
    BaseAddress = aMemory.readShort (0x08);
  else if ((stricmp(name, "lpt2")==0))
    BaseAddress = aMemory.readShort (0x0a);
  else if ((stricmp(name, "lpt3")==0))
    BaseAddress = aMemory.readShort (0x0c);
  else if ((stricmp(name, "lpt4")==0))
    BaseAddress = aMemory.readShort (0x0e);
} .
```

Die Funktionen zum Lesen und Schreiben der Datenleitungen verwenden die Klasse `IOPort` des UNIIO-Pakets:

```
void lptPort::writeData(const int data)
{
  if (BaseAddress==0)
    return;
  IOPort P(BaseAddress+DATA);
  P.writeChar(data);
}
int lptPort::readData()
{
  if (BaseAddress==0)
    return -1;
  IOPort P(BaseAddress+DATA);
  return (unsigned char) P.readChar();
} .
```

Aus `writeData` und `readData` werden die Funktionen zur Bitmanipulation zusammengesetzt:

```
void lptPort::setBit(int n)
  { writeData(readData() | Bit(n)); }
void lptPort::resetBit(int n)
  { writeData(readData() & ~Bit(n)); } .
```

Die Verwendung der Klasse `lptPort` zusammen mit den Multimedia-Timer-Funktionen (vgl. 7.3.6) in einem kurzen Programm, dem die Anzahl der Schrittmotorschritte in der Kommandozeile übergeben wird, könnte dann so aussehen:

```
#include "mmsystem.h"
#include "lpt.h"
```

```
...
const int Delay = 1;
static UINT uTimer = 0;
static int running = 1;
lptPort pp("LPT1");

void CALLBACK Steps ( UINT ID, UINT, DWORD steps, DWORD, DWORD )
{
  static unsigned short n = 0;
  if ((n%2)==0)
    pp.setBit(0);
  else
    pp.resetBit(0);
  if (++n >= 2*steps)
    running = 0;
}

int main(int argc, char *argv[])
{
  int steps = 10;
  if (argc>1)
    steps = atoi(argv[1]);
  if (steps==0)
    return -1;
  pp.resetBit(0);
  if (steps<0) {
    pp.setBit(1);
    steps = -steps;
  }
  else
    pp.resetBit(1);
  timeBeginPeriod(1);
  uTimer = timeSetEvent ( Delay, 1, Steps, steps, TIME_PERIODIC );
  while (running)
    ;
  timeKillEvent ( uTimer );
  timeEndPeriod(1);
  Sleep (100);
  return 0;
}
```

`timeBeginPeriod` und `timeEndPeriod` klammern die Verwendung der Multimedia-Timer-Ressource ein, der Parameter ist die gewünschte Zeitauflösung. Mit `Delay` = 1 wird ein Schritt-Takt von 500 Hz erreicht, passend für den Betrieb der meisten Schrittmotoren.

7.4.3 Servo-Ansteuerung: Multimedia-Timer und Performance-Counter

Ähnlich wie Schrittmotoren können auch Servos über das Druckerport betrieben werden. Wie in Abschnitt 2.4 beschrieben wurde, werden zur Ansteuerung gut definierte Impulssequenzen benötigt, bei denen die Folgefrequenz fest ist und die Impulslänge genau eingestellt werden kann. Die feste Folgefrequenz kann durch den Multimedia-Timer realisiert werden, die Pulslänge mit dem Performance-Counter gemessen werden.

Ein einzelner Impuls mit genau einstellbarer Länge wird bei jedem Aufruf der Callback-Funktion `Pulse` erzeugt:

```
short ACTUAL = 30;

void CALLBACK Pulse ( UINT ID, UINT, DWORD, DWORD, DWORD )
{
  LARGE_INTEGER f, t1, t2;
  QueryPerformanceFrequency(&f);
  unsigned short Inc = f.QuadPart/20000*ACTUAL;
  QueryPerformanceCounter(&t1);
  t1.QuadPart += Inc;
  pp.setBit(0);
  do {
    QueryPerformanceCounter(&t2);
    } while (t2.QuadPart<t1.QuadPart);
  pp.resetBit(0);
}
```

Die Voreinstellung für die Pulslänge ist 1.5 ms (vgl. Abschnitt 2.4).

Im Hauptprogramm wird der Multimedia-Timer gestartet, hier mit einer Zeitkonstante von 20 ms. In einer Schleife wird durch Tastatureingaben die Pulslänge, damit die Winkelstellung des Servos, verändert.

```
const short MIN = 18, MAX = 42;
timeBeginPeriod (20);
uTimer = timeSetEvent ( 20, 20, Pulse, 0, TIME_PERIODIC );
while ((c=_getch())!='e')
  switch (c) {
    case 'a': ACTUAL = MIN;
      break;
    case 's': ACTUAL = (ACTUAL<=MIN) ? MIN : (ACTUAL-1);
      break;
    case 'd': ACTUAL = (ACTUAL>=MAX) ? MAX : (ACTUAL+1);
      break;
    case 'f': ACTUAL = MAX;
      break;
```

```
        default:
          ;
      }
    timeKillEvent ( uTimer );
    timeEndPeriod (20);
```

Anders als in diesem Testprogramm wird man in einem arbeitsfähigen Steuerprogramm die Pulslängen jeweils problemangepasst einstellen und nur eine begrenzte Anzahl von Steuerimpulsen ausgeben (vgl. Abschnitt 7.4.2).

7.4.4 Enhanced Parallel Port (EPP) und Extended Capability Port (ECP)

Durch verschiedene Verbesserungen wurde in den vergangenen Jahren die Leistungsfähigkeit der Druckerschnittstelle deutlich erhöht. Die von unterschiedlichen Herstellern eingeführten Veränderungen wurden 1994 vom IEEE[41] vereinheitlicht und in einer neuen Norm für die Druckerschnittstelle – IEEE 1284 – verabschiedet. Praktisch alle Hersteller halten sich inzwischen an diesen Standard. Die Norm enthält die Druckerschnittstelle alter Art als Basisbetriebsart, ist also voll rückwärtskompatibel, daneben sind verschiedene bidirektionale Betriebsarten (u. a. EPP und ECP) definiert, die einen schnellen Datenaustausch (bis zu 2 MByte/sec) mit Peripheriegeräten wie Scannern oder externen Laufwerken ermöglichen. Neben den unterschiedlichen Betriebsarten sind im Standard IEEE 1284 auch die elektrischen Daten der Schnittstelle wie Steckerbelegung, Kabelart, maximale Kabellänge (10 m) usw. genauestens spezifiziert.

Eine gute und ausführliche Beschreibung der Norm bietet die Webseite von Warp Nine Engineering [39], eines Herstellers von Peripherie-Bausteinen für die IEEE 1284 (eine partielle Kopie davon liegt auf dem lokalen Server [40]). Hardware-Beschreibungen sind in den Datenblättern für die entsprechenden Schnittstellen-ICs enthalten (z. B. für die ICs TL16PIR552 – einem Baustein mit zwei seriellen und einem parallen Port oder SN74LVC161284 – einem IEEE-1284-kompatiblen Treiberbaustein).

7.4.5 Der programmierbare Parallel-E/A-Baustein 8255

Auf vielen Steckkarten, die parallele Ein/Ausgabeleitungen zur Verfügung stellen und/oder über solche statischen Leitungen andere Bausteine betreiben (D/A-, A/D-Wandler, Zähler, Relais), wird (besser: wurde) zu diesem Zweck der Intel-Schaltkreis 8255 A eingesetzt. Er besitzt 24 programmierbare E/A-Leitungen, die in drei Gruppen zu je 8 unterteilt sind (Ports A, B, C). Port C lässt sich weiter in 2 Gruppen zu je 4 aufteilen (CH, CL). Für jede der Gruppen kann (muss) die Betriebsart (Eingabe oder Ausgabe) definiert werden, so dass je nach Programmierung 0, 4, 8 $\cdots$ 24 Eingabe- und entsprechend 24, 20, 16 $\cdots$ 0 Ausgabeleitungen vorhanden sind. Der Baustein belegt 4 aufeinanderfolgende Adressen im E/A-Adressraum, die Basisadresse ist Port A zugeteilt, Basis+1 Port B, Basis+2 Port

[41] Institute of Electrical and Electronics Engineers [38].

C, auf Basisadresse+3 wird das Steuerbyte geschrieben, das entweder Betriebsart und Betriebsmodus festlegt oder ein Einzelbitkommando enthält. Die genaue Beschaltung der E/A-Leitungen (Steckerart und -belegung etc.) variiert von Karte zu Karte und muss im Einzelfall der Beschreibung entnommen werden.

Aufbau des Steuerbytes für den Parallelbaustein 8255 A (Betriebsart und -modus):

BIT	7	6	5	4	3	2	1	0
	1	M2	M1	A	CH	M0	B	CL

BIT 7 = 1 legt fest, dass mit diesem Kommandobyte Betriebsart und Betriebsmodus festgelegt werden sollen.

M2, M1 definieren den Betriebsmodus für Port A und CH,

M0 den für Port B und CL, diese 3 Bits sollten auf 0 gesetzt werden, dies bedeutet normale Ein- oder Ausgabe. Die anderen möglichen Betriebsmodi (automatischer Quittungs- oder Interruptbetrieb) sind hardwaremäßig meist nicht vorgesehen.

A, CH, B, CL definieren die Betriebsart für die einzelnen Gruppen, ein auf 0 gesetztes Bit bedeutet Ausgabe, ein auf 1 gesetztes Eingabe.

Das Steuerbyte 83H würde mithin die Gruppen A und CH als Ausgabe-, die Gruppen B und CL als Eingabeleitungen festlegen.

Wie beim Druckerport kann auch beim 8255 der aktuelle Status der Ausgabeleitungen durch einen Input-Befehl gelesen werden. Dies ist wichtig, wenn es darauf ankommt, nur einzelne Leitungen (Bits) anzusprechen, ohne die übrigen eines Ports oder einer Gruppe zu verändern.

Für Port C gibt es beim 8255 eine zusätzliche Möglichkeit, einzelne Bits gezielt anzusprechen, das Einzelbitkommando, bei dem das Steuerbyte (Basisadresse+3) wie folgt aufgebaut sein muss:

BIT	7	6	5	4	3	2	1	0
	0				N2	N1	N0	W

BIT 7 = 0 ist das Zeichen für ein Einzelbitkommando.

N2, N1, N0 adressieren das Bit in Port C, das auf den Wert von

W gesetzt wird.

0x01 würde Bit 0 auf 1 setzen, 0x06 Bit 3 auf 0, 0x0f Bit 7 auf 1.

Die Programmierung des Bausteins 8255 soll am Beispiel einer Klasse für den Betrieb eines externen 14-Bit Puls-A/D-Wandlers an einer älteren Parallel-E/A-Karte näher erläutert werden.

In der Header-Datei ist als Kommentar ein schematisches Blockdiagramm vorangestellt, das auch die Bit-Zuordnungen der 8-Bit-Ports der beiden 8255 auf der Karte zu den Anschlussleitungen der Analog-Digital-Wandler definiert (mit den beiden 8255-Bausteinen auf der Karte ist der gleichzeitige Betrieb von zwei externen A/D-Wandlern möglich).

```
/*-------------------------------------------------------------
               PC BUS (ADDRESS, DATA, CONTROL)
 -----+     +---------------+     +---------------+     +-----
 +---------------+     +---------------+     +---------------+
 |     PIO  1    |     |               |     |     PIO  2    |
 |      8255     |     |  TIMER 8253   |     |      8255     |
 |     24 BIT    |     |               |     |     24 BIT    |
 +---------------+     +---------------+     +---------------+
   |A|  |B|  |C|                               |A|  |B|  |C|

 +---------------+                           +---------------+
 |     ADC 1     |                           |     ADC 2     |
 +---------------+                           +---------------+
         |                                           |

 A0..B5 : DATA      ...  from ADC  ...  low active (1)
 B6     : OVERFLOW  ...  from ADC  ...  low active
 B7     : READY     ...  from ADC  ...  low active
 C0     : CLEAR     ...  to ADC    ...  low active
 C1     : ADC RUN   ...  to ADC    ...  low active        */
```

Die Datenbits des Analog-Digital-Wandler-Ausgangs sind in logischer Abfolge ihrer Wertigkeit entsprechend mit den 8 Bits von Port A und 6 Bits von Port B verbunden, die Statusausgänge des ADC mit den restlichen 2 Bits von Port B , 2 Bitausgänge von Port C steuern den ADC (realisiert wird dies durch ein speziell für den Anwendungszweck gefertigtes Verbindungskabel).

In der Klassendefinition werden nach der Vereinbarung von Variablen und Hilfsfunktionen die Portadressen und Konfigurationskommandos für die 8255er sowie die Bitmasken für die Ports festgelegt. Gut dokumentieren und kapseln lässt sich dies mit Aufzählungstypen, in denen das Makro `BIT()` zum Setzen einzelner Bits verwendet wird.

```
#define BIT(b) (1<<b)

class PIO8255 : CInOut {
```

```
private:
  unsigned short BaseAddress;
  unsigned short ADDR (unsigned short offset = 0) {
      return (BaseAddress+offset);
  };
  enum PortOffsets { LOPORT, HIPORT, CMDPORT, CONFPORT };
  enum PortConfiguration {
      ALLOUTPUT = BIT(7),
      AINPUT    = BIT(7) | BIT(4),
      BINPUT    = BIT(7) | BIT(1),
      CLINPUT   = BIT(7) | BIT(0),
      CHINPUT   = BIT(7) | BIT(3),
      ADCREAD   = AINPUT | BINPUT
  };
  enum ADCStatus {
      NOTOVER   = BIT(6),
      NOTREADY  = BIT(7),
  };
  enum ADCCommand {
      CLEARCMD  = 0,
      RUNCMD    = BIT(0),
      STOPCMD   = BIT(0) | BIT(1)
  };
  void Acknowledge () {
      IOPort CMD ( ADDR(CMDPORT) );
      CMD.writeChar ( CLEARCMD );
      CMD.writeChar ( RUNCMD );
  };
```

Mit der Funktion `Acknowledge()` wird ein aus dem A/D-Wandler gelesener Digitalwert quittiert.

Öffentlich, d. h. für den Benutzer der Klasse verwendbar sind schließlich nur die beiden Methoden `DataAvailable()` und `GetData()`. Für den Normalbetrieb der Analog-Digital-Wandler ist dies ausreichend.

```
public:
  PIO8255 (unsigned short basead = 0x1b0) {
      BaseAddress = basead;
      IOPort CONF ( ADDR(CONFPORT) );
      IOPort CMD ( ADDR(CMDPORT) );
      CONF.writeChar ( ADCREAD );
      CMD.writeChar ( STOPCMD );
      Acknowledge();
  };
```

```
    bool DataAvailable () {
        IOPort HI ( ADDR(HIPORT) );
        return ( ( HI.readChar() & NOTREADY ) == 0 );
    };
    unsigned short GetData () {
        unsigned short data;
        IOPort HI ( ADDR(HIPORT) );
        IOPort LO ( ADDR(LOPORT) );
        data = ( (HI.readChar() | NOTREADY)<<8 ) + LO.readChar();
        Acknowledge();
        return ~data;
    };
};
```

7.5 Der IEC-Bus

Die in der vorhergehenden Kapiteln beschriebenen Schnittstellen (parallel, seriell) werden im wesentlichen für 2-Punkt-Verbindungen benutzt (PC↔Plotter, PC↔Drucker). Diese Verbindungstechnik wird sehr aufwendig, wenn eine größere Anzahl von Messgeräten eingebunden werden soll, da der PC für jedes Messgerät eine individuelle Schnittstelle zur Verfügung stellen muss.

Im Gegensatz zu den 2-Punkt-Verbindungen zeichnen sich Bussysteme dadurch aus, dass eine Vielzahl von Geräten an eine Schnittstelle angeschlossen werden kann, und über den Bus eine Kommunikation mit dem PC wie auch zwischen den angeschlossenen Geräten möglich ist.

7.5.1 Grundlagen

Das Bussystem, das sich auf dem Messgerätesektor seit einigen Jahren weitgehend durchgesetzt hat, ist der IEC-Bus. Dieser sehr gut genormte Schnittstellenbus[42] besteht aus 8 Daten-, 3 Handshake- und 5 allgemeinen Steuerleitungen (Abb. 62).

Der Bus wird in negativer Logik mit TTL-Spannungen betrieben, der aktive Leitungszustand bzw. logisch 1 entspricht somit ca. 0 V, inaktiv oder logisch 0 dagegen ca. 5 V. Die Signalausgänge sind als 'offene Kollektor' Ausgänge realisiert, durch diese 'Wired-Or'-Verknüpfung wird sichergestellt, dass auch dann keine Buskonflikte auftreten, wenn mehrere Geräte gleichzeitig auf eine bestimmte Busleitung zugreifen. Andererseits kann ein einzelnes Gerät den Aktiv-Status einer Leitung erzwingen.

Gegenüber einem Rechnerbus weist der IEC-Bus einige Unterschiede auf:

[42] Die international gültigen Schnittstellen-Normen sind IEEE 488 (USA) und IEC 66.22 (Europa). Für den IEC-Bus sind auch die Bezeichnungen IEEE-Bus (sprich: ai-triple-i), HPIB (Hewlett Packard Interface Bus) und GPIB (General Purpose Interface Bus) gebräuchlich.

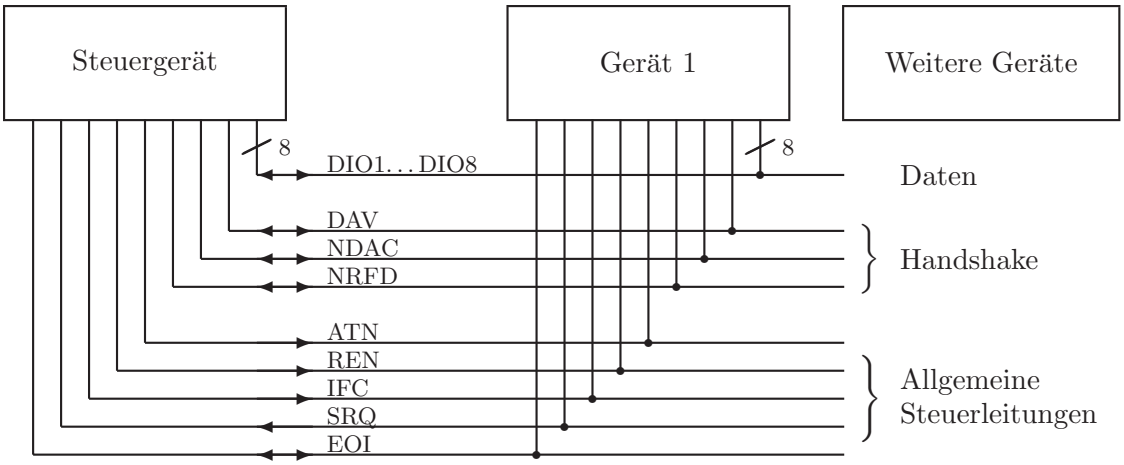


Abbildung 62: Die 16 Signalleitungen des IEC-Bus.

Die Datenübertragung erfolgt asynchron, d. h. sie wird nicht — wie beim PC-Bus — durch ein zentrales Taktsignal synchronisiert, sondern durch ein Quittungsverfahren mit den 3 Handshakeleitungen. Das Signal für gültige Daten (DAV: Data Valid) wird vom jeweiligen Sprechergerät nur dann aktiviert, wenn alle Hörergeräte am Bus ihre Bereitschaft durch das deaktivierte NRFD-Signal (Not Ready For Data) anzeigen. Die Daten bleiben danach so lange gültig, bis alle Hörergeräte das NDAC-Signal (Not Data Accepted) deaktiviert und damit gemeldet haben, dass die Daten empfangen wurden. Die Datenübertragungsgeschwindigkeit richtet sich also dynamisch nach dem jeweils langsamsten aktiven Gerät am Bus. Dennoch sind — bei hinreichend schnellen Teilnehmern — maximale Übertragungsraten von ca. 1 MByte/sec möglich.

Die 8 Datenleitungen (DIO1...DIO8) werden sowohl für Daten als auch für Kommandos benutzt, die Unterscheidung erfolgt durch die Steuerleitung Attention (ATN). Nur ein Gerät am Bus — als Steuergerät (Controller) konfiguriert — darf die Attention-Leitung bedienen und somit Kommandos erteilen. Ein Teil der Kommandos dient dazu, andere Geräte am Bus als Sprecher- (Talker) oder Hörer-Geräte (Listener) zu adressieren. Durch diese Adressierkommandos wird eine Datenübertragung zwischen einem Sprecher und einem oder mehreren Hörern eingeleitet. Insgesamt sind 31 Hörer- und 31 Sprecheradressen in einem IEC-Bus-System möglich, die übrigen Kommandobytes sind für weitere Befehle reserviert. Unter anderem ist eine weitergehende Adressunterteilung über Sekundäradressen möglich.

Die Funktion der weiteren Steuerleitungen:

REN (Remote Enable) schaltet die lokale Bedienungsmöglichkeit der angeschlossenen Geräte ab.

IFC (Interface Clear) bringt das Schnittstellensystem in einen definierten Anfangszustand.

SRQ (Service Request) wird von einem angeschlossenen Gerät aktiviert, wenn es eine Bedienung wünscht (z. B. am Ende einer Messung zum Datentransfer). Die dazu nötige Unterbrechung wird vom Steuergerät veranlasst, erfolgt daher nicht unbedingt prompt. Die Reaktion des Steuergeräts muss mit einer Umfrage (Serial Poll, Parallel Poll) beginnen, um festzustellen, welches Gerät die SRQ-Leitung aktiviert hat.

EOI (End Or Identify) hat zwei Funktionen: Zum einen zeigt der gerade aktive Sprecher damit das Ende des Datentransfers an. Zum anderen wird EOI zusammen mit ATN vom Controller benutzt, um eine Parallelabfrage (Parallel Poll) anzuordnen.

Für weitere technische Details sei an dieser Stelle auf die weiterführende Literatur zum IEC-Bus verwiesen [41, 42].

7.5.2 Datenformat

Die Datenübertragung auf dem IEC-Bus erfolgt 'bit-parallel' und 'byte-seriell'. Hinsichtlich der Länge des Datenstroms und der Codierung der einzelnen Bytes bestehen keine Einschränkungen. Daher muss die Art der Datencodierung jeweils zwischen Talker- und Listener-Gerät vereinbart werden. Da aber bei praktisch allen Peripheriegeräten das Datenformat für den IEC-Bus nicht veränderbar ist, bedeutet 'vereinbaren', dass sich der PC als flexibelstes Gerät am Bus auf das jeweilige Datenformat einstellen muss. Zwei Codierungsarten werden überwiegend verwendet:

- Binäre Codierung dort, wo große Datenmengen möglichst schnell übertragen werden sollen. Einige Digitalspeicheroszilloskope übertragen ihre Daten auf diese Weise — bei einer Auflösung von 8 Bit wird für einen Datenpunkt nur 1 Byte benötigt.
- ASCII-Codierung dort, wo es eher auf Sicherheit und Verständlichkeit ankommt und die Geschwindigkeit nicht im Vordergrund steht. Die Daten werden als Text übertragen, somit ist eine sofortige Kontrolle möglich. Fast alle Digitalmultimeter benutzen diese Codierungsart sowohl für ihre Programmierung als auch für die Messdaten. Da jeweils nur ein Messwert übertragen wird, werden keine hohen Geschwindigkeiten benötigt.

7.5.3 Programmierung

PC-Karten für den IEC-Bus enthalten im allgemeinen einen intelligenten Schnittstellenbaustein, der einen großen Teil des Busmanagements (insbesondere den Quittungsablauf) selbständig erledigt. Meist können die Karten sowohl als Controller (überwiegende Betriebsart im PC) wie auch als Talker/Listener konfiguriert werden.

Fast alle Kartenhersteller (jedenfalls die teureren) liefern Softwarebibliotheken zum Betrieb der Karte mit, üblich ist zumindest eine C-Bibliothek. Der Datenaustausch mit einem

externen Gerät erfolgt dann hochsprachlich analog zum Dateizugriff nach dem Schema:

1. Logische Verbindung zum externen Gerät herstellen (*open*),
2. Daten lesen (*read*) oder schreiben (*write*),
3. Verbindung zum externen Gerät schließen (*close*).

Beispiel: Hewlett-Packard, neben National Instruments und Keithley einer der wichtigeren Hersteller für IEC-Bus-Karten, liefert zum Betrieb der Karten die *HP Standard Instrument Control Library* mit, eine Bibliothek mit Schnittstellen für Basic und C. Die Software[43] muß zunächst auf dem PC installiert werden, anschließend wird die Schnittstellenkarte logisch eingebunden, d. h. unter einem symbolischen Namen werden die physikalischen Daten der Karte (E/A-Adresse usw.) vermerkt (zuständiges Programm: `iocfg32.exe`). Der bei der Konfiguration festgelegte Name wird – zusammen mit der Adresse des externen Geräts – beim Verbindungsaufbau benötigt. Die Bibliotheksfunktionen werden in C bzw. C++ durch die Header-Datei `sicl.h` bekanntgemacht und aus `sicl32.lib` mit dem Linker ins Programm eingebunden. Aus den Routinen in `sicl32.lib` erfolgen Zugriffe auf die zentrale System-DLL `sicl32.dll`, die das eigentliche Ein/Ausgabe-Management abwickelt. Durch diese Softwarestruktur ist gewährleistet, daß alle Zugriffe auf die IEC-Bus-Karte(n) im System über eine zentrale Stelle laufen; die verantwortliche DLL ist 'Multi-Thread'-fähig, somit können mehrere Programme gleichzeitig mit dem Bus arbeiten.

Unter anderem stellt SICL die folgenden Definitionen und Funktionen für den Betrieb von Geräten bereit (aus `sicl.h`):

```
typedef int INST;
INST iopen (char *addr);
int iclose (INST id);  .
```

Auf ein Geräte wird über eine 'Instrument-Handle' zugegriffen, die in Typ und Verwendungsart in etwa der 'File-Handle' in C entspricht. `iopen()` eröffnet die Verbindung zum Gerät, `addr` enthält Kartenname und Geräteadresse, also z. B. "hpib,7".

Die Funktionen für formatiertes Schreiben und Lesen entsprechen den C-Bibliotheksfunktionen `fprintf()` und `fscanf()`:

```
int iprintf (INST id, const char *fmt, ...);
int iscanf (INST id, const char *fmt, ...);  .
```

Der Rückgabewert enthält die Anzahl der tatsächlich konvertierten Argumente.

Zum unformatierten Schreiben und Lesen aus einem oder in ein Bytefeld `buf` der Länge `datalen` bzw. `bufsize` sind

```
int iwrite (INST id, char *buf, unsigned long datalen,
            int endi, unsigned long *actual);
int iread (INST id, char *buf, unsigned long bufsize,
```

[43] Vorsicht bei Windows NT: Fehlerfrei unter NT 4.0 arbeiten nur neuere Versionen (etwa ab Rev. F.01.02 1997).

```
                    int *reason, unsigned long *actual);
```

zuständig, `endi` legt die Art des Übertragungsendes fest (= 0: ohne **EOI**, ≠ 0: mit **EOI**), `actual` gibt die Zahl der tatsächlich übertragenen Bytes an und `reason` den Grund für Erfolg oder Mißerfolg. Die Rückgabewerte sind C-üblich Null bei Erfolg, ungleich Null bei Mißerfolg.

Ein Programm, das ein Digitalvoltmeter auf einen bestimmten Betriebszustand setzt und dann einen Meßwert liest, könnte die folgenden Zeilen enthalten:

```
#include "c:/hp/siclnt/c/sicl.h"
...
INST DVM = iopen("hpib,7");
iprintf (DVM, "LOVAT0\n");
iscanf (DVM, "%lg", &Voltage);
iclose(DVM);  .
```

Die Details der Funktionen sowie die Vielzahl weiterer SICL-Funktionen sind in den bei den HP-Karten mitgelieferten umfangreichen Handbüchern [43, 44] ausführlich beschrieben.

7.6 Universal Serial Bus (USB)

Von einem Herstellerkonsortium bestehend aus Compaq, Digital Equipment Corporation, IBM PC Company, Intel, Microsoft, NEC und Northern Telecom wurde nach mehrjähriger Vorarbeit im Januar 1996 eine neue Schnittstellennorm für den PC vorgeschlagen [45], der *Universal Serial Bus* (USB). Alle neueren PCs enthalten inzwischen diese Schnittstelle (gekennzeichnet durch das nebenstehende Logo), die Zahl der damit ausgestatteten Peripheriegeräte wächst laufend [46]. Seit einiger Zeit gibt es auch Messgerätehersteller, die diesen Bus verwenden. In allen neueren PC-Betriebssystemen ist die Unterstützung für den USB integriert, spezialisierte Treiber werden von den Geräteherstellern in praktisch allen Fällen mitgeliefert, Eigenprogrammierung ist daher nicht erforderlich.

Von den genannten Herstellern ist der USB so universell konzipiert, dass er langfristig alle bisherigen Standardschnittstellen überflüssig machen kann. Im Normvorschlag sind verschiedene Geschwindigkeitsbereiche vorgesehen, so dass auf dem Bussystem langsame und schnelle Peripheriegeräte koexistieren können, ohne dass überall ein hoher Aufwand für schnellen Schnittstellenbetrieb nötig ist. Einen Überblick über die Geschwindigkeitsbereiche des USB gibt die Zusammenstellung in Abbildung 63. Vor kurzem wurde die Norm für den Hochgeschwindigkeitsbereich verabschiedet (USB 2.0), neuere Betriebssysteme unterstützen inzwischen auch diesen Bereich.

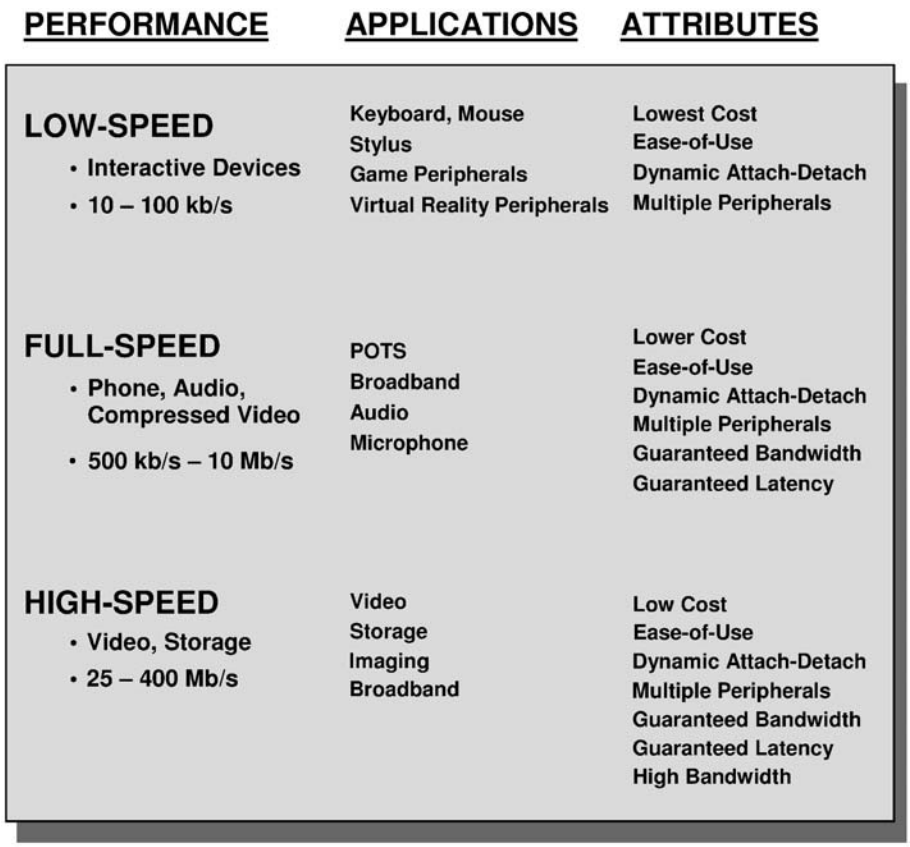


Abbildung 63: Geschwindigkeitsbereiche für den USB, Anwendungen und typische Eigenschaften (Stand USB 2.0, 2002).

Das Konzept des USB sieht eine hierarchisch vernetzte Peripheriestruktur vor, in der die einzelnen Geräte durch logische Adressen angesprochen werden.

Ähnlich wie beim Twisted-Pair-Ethernet ist die logische Busstruktur physikalisch durch Zweipunktverbindungen realisiert, die über Verteiler (Hubs) verbunden sind. An einem Root-Hub (in der Regel im PC) können bis zu 127 Geräte angeschlossen werden. Das Prinzip ist in Abbildung 64 skizziert.

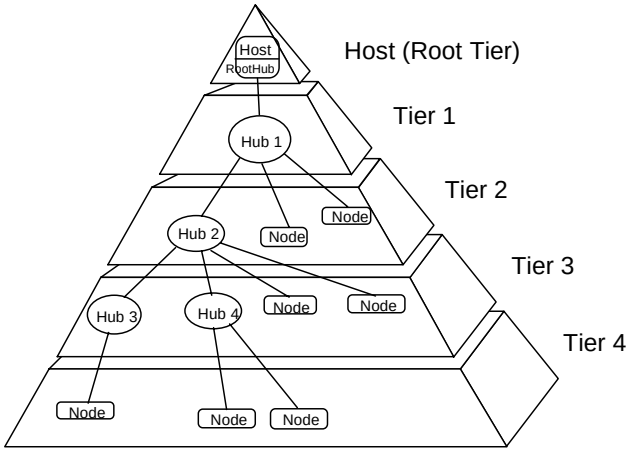


Abbildung 64: Vernetzungsstruktur eines USB-Systems.

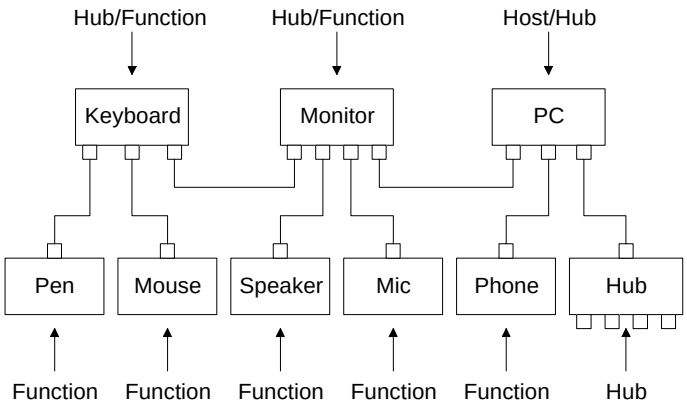


Abbildung 65: PC-System, das über USB-Verbindungen betrieben wird.

In den Endgeräten kann die Hub-Funktionalität mit integriert sein, somit kann ein einfaches USB-System auch ohne abgesetzte Hubs aufgebaut werden. Eine mögliche Realisierung für ein PC-System zeigt Abbildung 65: Die Hauptperipheriegeräte wie Monitor und Keyboard fungieren gleichzeitig als USB-Hub, an den jeweils weitere Geräte angeschlossen sind.

Als serielles Bussystem benötigt der USB nur relativ einfache Kabelverbindungen; für den Datentransfer wird eine verdrillte Zweidrahtleitung verwendet, daneben wird noch die Versorgungsspannung von 5 V über die Buskabel geführt (Abbildung 66). Dadurch können Geräte mit geringem Stromverbrauch auch ohne eigenes Netzteil am USB betrieben werden.

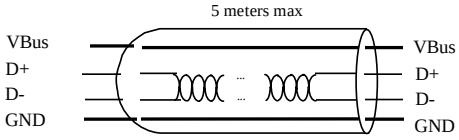


Abbildung 66: USB-Kabel: Verdrillte Datenleitungen und zwei Leitungen für die Versorgungsspannung reichen aus.

In der Spezifikation weiterhin vorgesehen ist eine *Hot-Plug*-Fähigkeit: Peripheriegeräte können während des normalen Rechnerbetriebs hinzugefügt oder entfernt werden, der lästige und zeitraubende Neustart des Systems entfällt. Die Geräte werden automatisch erkannt, der zugehörige Treiber wird geladen bzw. aktiviert oder nach dem Entfernen eines Geräts wieder deaktiviert.

Für den Bereich der Datenerfassung und Experimentsteuerung weist der USB einige interessante Vorteile auf:

- Als breit akzeptierter Standard ist die Schnittstelle überall verfügbar (Notebooks). Der Einbau von speziellen Schnittstellenkarten entfällt.
- Die notwendigen Komponenten sind billig herzustellen (einfache standardisierte Kabel, intelligente Standardperipheriebaussteine).

- Einfache Systeme lassen sich aus wenigen ICs zusammenbauen, oft reicht *ein* intelligenter Peripheriebaustein aus, die Stromversorgung wird vom Bus geliefert.
- Da bis zu 127 Geräte am Bus betrieben werden können, muss man nicht mehr möglichst viele Teilfunktionen zusammenfassen, um Platz zu sparen (Multi-IO-Karten), sondern kann die Funktionen getrennt und damit flexibler realisieren.

Für die Programmierung werden jedoch immer USB-konforme, ins Betriebssystem eingebundene Gerätetreiber nötig sein, wegen der komplexen Fähigkeiten des USB verbietet sich ein direkter Zugriff auf Geräte.

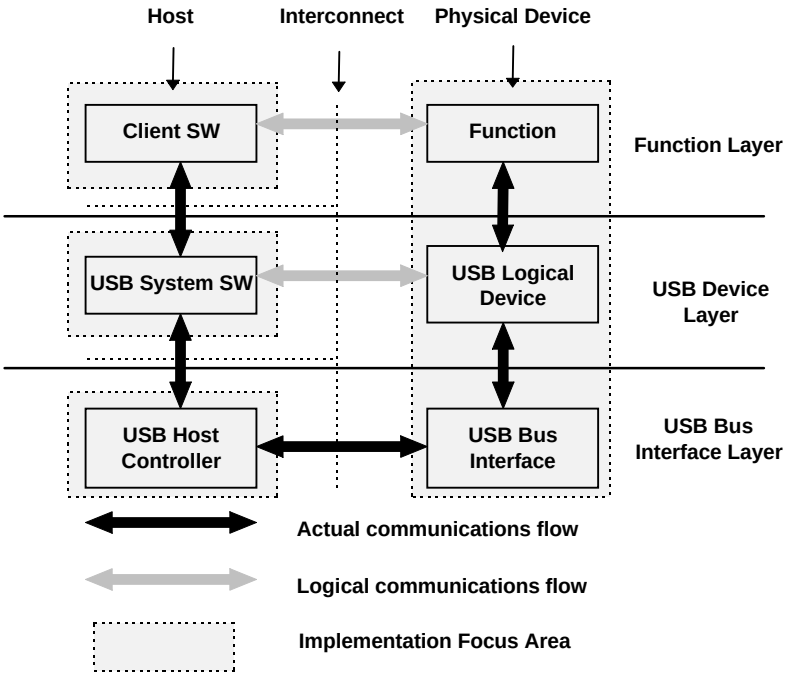


Abbildung 67: Logischer und physikalischer Datenfluss in einem USB-System.

Der Datenfluss in einem USB-System ist in Abbildung 67 dargestellt. Im intelligenten Peripheriegerät wird man in der Regel nur die oberste Ebene (Function Layer) implementieren müssen, die beiden unteren Ebenen können aus vorgefertigten Moduln entnommen werden. Ähnlich ist es im Rechnersystem, auch dort wird die Software auf fertigen Moduln (DLLs und Gerätetreiber) aufsetzen. Konkret bedeutet das, dass bei kommerziellen Peripheriegeräten im Anwenderprogramm die Funktionen der mitgelieferten DLLs benutzt werden, mithin nur der logische Datenfluss der obersten Ebene programmiert wird, bei Eigenbaugeräten wird man allerdings DLL, Gerätetreiber und (zumindest) die oberste Softwareebene im Peripheriegerät selbst realisieren.

Die gesamten technischen Spezifikationen des Universal Serial Bus sind sehr detailliert in [47] zusammengestellt, alle Abbildungen des Kapitels sind dieser Dokumentation entnommen.

7.7 TCP/IP-Programmierung

Neben der ständig wachsenden Zahl der rein windowsorientierten Kommunikationsmöglichkeiten (DDE, NetDDE, OLE1, OLE2, COM, DCOM, ActiveX, ...), steht – wenn TCP/IP als Protokoll installiert ist – unter den Windows-Systemen auch die weitgehend systemunabhängige Socket-Schnittstelle zum Informationsaustausch zwischen Prozessen auf einem oder auf verschiedenen vernetzten Rechnern (Internet) zur Verfügung. Über diese Schnittstelle ist auch eine Verbindung mit Rechnern unter anderen Betriebssystemen einfach realisierbar. Darüber hinaus sind seit einiger Zeit verschiedene Microcontroller auf dem Markt, die eine Ethernet-Schnittstelle und die zugehörige TCP/IP-Software enthalten und damit den Bau einfacher und billiger Messsysteme mit Internet-Anschluss ermöglichen.

7.7.1 Sockets und Ports

Ein Socket ist der Endpunkt einer Netzwerkverbindung unter dem TCP/IP-Protokoll; diese Schnittstelle kann überall dort benutzt werden, wo TCP/IP als Netzwerkprotokoll installiert ist. Die Kommunikation über Sockets kann sowohl zwischen Programmen innerhalb eines Rechners wie auch – das ist der üblichere Fall – zwischen unterschiedlichen Rechnern ablaufen.

Der Verbindungsaufbau erfolgt (bei TCP-Verbindungen, auf die wir uns hier beschränken wollen) nach einem Client-Server-Prinzip: ein Server-Socket wartet am Netzwerk (`Listen`), ein Client-Socket betreibt den Verbindungsaufbau (`Connect`), der Server-Socket nimmt die Verbindung an (`Accept`). Über die Verbindung werden Daten mit `Send` und `Receive` ausgetauscht, die Verbindung wird schließlich von einem der Partner mit `Close` beendet.

Die Art der Verwendung bzw. das Subprotokoll der Verbindung wird durch die Port-Nummer des Server-Sockets spezifiziert. Dadurch können unterschiedliche Verbindungstypen (TELNET, FTP, WWW, ...) quasigleichzeitig auf eine physikalische Netzschnittstelle zugreifen – soweit dafür ein Server-Programm vorgehalten wird. So hat beispielsweise ein TELNET-Server die Port-Nummer 23, ein SMTP-Server 25, ein WWW-Server 80. Der Client gibt beim Connect-Versuch die gewünschte Port-Nummer an, um mit dem zuständigen Server verbunden zu werden. Die üblichen Client-Programme machen dies ohne Zutun richtig – ein TELNET-Programm versucht einen Verbindungsaufbau mit Port 23 des Host-Rechners. Zu Testzwecken kann die Verbindung auch zu einer anderen Port-Nummer hergestellt werden (z. B. TELNET-Client ↪ SMTP-Server, um das Protokoll zu studieren).

Generell ist vorgesehen, dass von einem Server mehrere Verbindungen vom gleichen Typ gemanagt werden können, dazu wird der Listener-Socket bei der Verbindungsannahme jeweils dupliziert und die Datenübertragung wird auf dem Duplikat abgewickelt.

Die Liste der festgelegten Port-Nummern ('well-known ports') ist in RFC1700 [48] veröffentlicht ('Request for Comments is a series of documents published by the Internet

Engineering Task Force [49] and cover a broad range of topics. The core topics are the Internet and the TCP/IP protocol suite.'). Die vom Rechnerbetriebssystem belegten Ports bzw. Dienste finden sich in der Datei `services` (bei UNIX-Systemen im Verzeichnis `/etc`, bei Windows-NT- oder -2000-Systemen im Verzeichnis `...\System32\drivers\etc`, bei Windows-95/98/ME-Systemen im Windows-Hauptverzeichnis). Bei der Programmierung von Socket-Verbindungen müssen Kollisionen von Port-Nummern mit 'well-knowns' vermieden werden (> 1024 im allgemeinen problemlos).

7.7.2 Socket-Server in C/C++

In C/C++ unter Linux oder Windows ist eine Socket-Verbindung meist mit relativ einfachen Anweisungen realisierbar, die zuständigen Funktionen ähneln den üblichen Datei-Ein/Ausgabe-Funktionen.

Bei einem Server-Socket wird die Verbindung mit `socket` und `bind` bereitgestellt, `listen` wartet dann auf ankommende Verbindungen, die mit `accept` angenommen werden. Dabei wird ein Duplikat des Sockets erstellt, mit dem die Datenübertragung durch die Funktionen `recv` und `send` abgewickelt wird. Der ursprüngliche Socket bleibt empfangsbereit, kann weitere Verbindungen abweisen, in eine Warteschlange stellen oder über weitere Duplikate sofort annehmen. Nach dem Ende der Verbindung wird das Socketduplikat mit `closesocket` geschlossen, der Server-Socket geht wieder in den normalen *Listen*-Zustand zurück.

Im folgenden Beispiel ist ein einfacher Echo-Server programmiert, ein Programm, das an einem bestimmten Port auf eine Verbindungsanforderung wartet und die ankommenden Daten reflektiert.

Port und Rechnername werden als Konstanten fest eingestellt:

```
const unsigned short PORT = 1234;
const char HOSTNAME[] = "localhost"; .
```

Unter Windows ist es zuallererst notwendig, das Socket-System zu initialisieren, das macht die Funktion `WSAStartup`:

```
WORD wVersionRequested = MAKEWORD( 1, 0 );
WSADATA wsaData;
WSAStartup( wVersionRequested, &wsaData ); .
```

Die Informationen über die Verbindungsart, Portnummer und IP-Adresse werden in einer Struktur von Typ `sockaddr_in` abgelegt, die Funktion `gethostbyname` übersetzt den Rechnernamen in eine IP-Adresse (lokal oder durch Anfrage bei einem Nameserver):

```
struct sockaddr_in ad, adcli;
int adlen = sizeof(sockaddr_in);
ad.sin_family = PF_INET;
ad.sin_port = htons(PORT);
```

```
struct hostent *he = gethostbyname(HOSTNAME);
if (he==0) {
  printf("Error: gethostbyname.\n");
return -1;
}
memcpy(&ad.sin_addr, he->h_addr, 4); .
```

Diese Struktur wird bei der Erstellung des Sockets in der Funktion `bind` als Parameter benötigt:

```
SOCKET S = socket(AF_INET, SOCK_STREAM, 0);
WSASetLastError(0);
if (bind(S, (sockaddr *) &ad, sizeof(ad))!=0) {
  printf("Bind Error: %d\n", WSAGetLastError());
  return -1;
  } .
```

In der folgenden Endlosschleife wird auf die Verbindungsanforderung durch einen Client-Socket gewartet, `listen` stoppt das Programm solange. `accept` nimmt die Verbindung auf dem Duplikat `T` an, in der Struktur `adcli` werden die Client-Daten abgelegt, man kann feststellen, mit wem man verbunden ist. Die zweite Schleife empfängt Daten, gibt sie auf dem Bildschirm aus und schickt sie wieder zurück, bis der Client die Verbindung beendet (`recv` liefert ein Ergebnis $<= 0$):

```
for (;;) {
  listen(S, 5);
  SOCKET T = accept(S, (sockaddr *) &adcli, &adlen);
  for (;;) {
    const int BUFSIZE = 1024;
    char buffer[BUFSIZE+1];
    int n = recv(T, buffer, BUFSIZE, 0);
    if (n<=0)
      break;
    buffer[n] = '\0';
    printf(buffer);
    send(T, buffer, strlen(buffer), 0);
  }
  closesocket(T);
}
return closesocket(S);  // never reached .
```

Nach der zweiten Schleife wird das Socket-Duplikat `T` mit `closesocket` geschlossen. Die äußere Endlosschleife kann nie verlassen werden, die letzte Zeile ist mithin nur ein Tribut an guten Programmierstil.

In einem Messsystem würde man statt der `printf`-Zeile eine geeignete Aktion implementieren, d. h. die Nachricht im `buffer` interpretieren, darauf reagieren, die Antwort im `buffer` ablegen.

Nach minimalen Änderungen ist das Programm auch unter Linux lauffähig:

- Alle Funktionen, die mit `WSA` beginnen, sind Windows-spezifisch, fallen unter Linux weg.
- Die Casts von `sockaddr_in` nach `sockaddr` sind unter Linux unnötig.
- Statt `closesocket` genügt unter Linux ein `close`.

7.7.3 Socket-Client in C/C++

Ein zu dem Socket-Server-Programm komplementäres Client-Programm, das Konsol-Eingaben an den Server schickt und die Antworten ausgibt, lässt sich durch kleine Änderungen realisieren.

Nach dem ungeänderten ersten Teil wird die Verbindung vom Client mit `connect` aufgebaut:

```
SOCKET S = socket(AF_INET, SOCK_STREAM, 0);
WSASetLastError(0);
if (connect(S, (sockaddr *) &ad, sizeof(ad))!=0) {
  printf("Connect Error: %d\n", WSAGetLastError());
  return -1;
  } .
```

Dann die Endlosschleife mit Tastatureingabe, Verschicken, Empfangen, Ausgabe:

```
for (;;) {
  const int BUFSIZE = 1024;
  char buffer[BUFSIZE+1];
  fgets(buffer, BUFSIZE, stdin);
  if ((buffer[0]=='e')||(buffer[0]=='E'))
    break;
  send(S, buffer, strlen(buffer), 0);
  int n = recv(S, buffer, BUFSIZE, 0);
  buffer[n] = '\0';
  printf(buffer);
}
return closesocket(S); .
```

7.7.4 Socket-Client in MATLAB/Java

Durch die Java-Integration ist es sehr einfach geworden, die TCP/IP-Kommunikation direkt in MATLAB zu formulieren. Das zuständige *Package* `java.net` wird importiert und es wird ein `Socket`-Objekt mit den benötigten Eigenschaften (Zielrechner, Port) formuliert:

```
import java.net.*
so = java.net.Socket('131.173.11.215', 1234);
...
V(i) = measure(so, C(i));
...
so.close; .
```

Die Messroutine `measure` übermittelt einen Vorgabewert `C(i)` an das Messsystem und liest einen Messwert `V(i)`:

```
function volt = measure(so, curr)
  sData = sprintf('da4 %d', curr);
  SocketIO(so, sData);
  rData = SocketIO(so, 'ad4');
  volt = str2double(rData); .
```

Die eigentliche Kommunikation wird von der Funktion `SocketIO` erledigt:

```
function recvStr = SocketIO(socket, sendStr)
  str = java.lang.String(sendStr);
  out = socket.getOutputStream;
  out.write(str.getBytes);
  in = socket.getInputStream;
  isr = java.io.InputStreamReader(in);
  ibr = java.io.BufferedReader(isr);
  recvStr = ibr.readLine; .
```

Der MATLAB-String wird in einen Java-String gewandelt, der `OutputStream` des Sockets wird etabliert, der Java-String wird verschickt. Auf den `InputStream` des Sockets wird ein `BufferedReader` aufgesetzt, mit dem eine ankommende Zeile gelesen wird.

8 Windows-Programmierung mit Visual C++

Einfache Messprogramme, Praktikumsanwendungen, Prototypprogramme, leicht veränderbare Programme oder solche mit nur kurzer Lebensdauer werden zweckmäßigerweise mit einer graphischen Messumgebung wie LabView bzw. VEE oder – wie in Kapitel 5 beschrieben – mit einem Numerikprogramm wie MATLAB programmiert. Für Aufgaben, die deutlich darüber hinaus gehen, beispielsweise die Erstellung von professionellen Messprogrammen für feste apparative Aufbauten, kann es sinnvoll sein, Windows-Programme komplett in einer einfachen Programmiersprache wie C, C++ oder Pascal (Delphi) zu erstellen.

Wegen des großen, praktisch unüberschaubaren Funktionenwildwuchses des Betriebssystems Windows – die Header-Datei *Windows.h* umfasst typischerweise 5000 bis 10000 Zeilen – ist es sehr aufwendig und fehleranfällig, umfangreiche Windows-Programme direkt in einer Sprache wie C zu schreiben. Sinnvoller ist es, sich geeigneter vorgefertigter Objekte und der zugehörigen objektorientierten Programmiersprache sowie einer darauf zugeschnittenen leistungsfähigen Programmierumgebung mit gutem *Online*-Hilfesystem zu bedienen. Am Beispiel von Visual C++[44] und der Klassenbibliothek der *Microsoft Foundation Classes* sollen hier einige der Möglichkeiten zur Erstellung von Windows-Messprogrammen diskutiert werden. Zur Realisierung werden die Hilfsmittel der Microsoft-Entwicklungsumgebung Visual Studio benutzt, die – zusammen mit den MFCs – in vielen neueren Büchern zur Windows-Programmierung (z. B. [50]) relativ gut dokumentiert sind.

> Man mag über die Firma denken wie man will;
> es gibt niemanden, der Windows besser kennt.

8.1 C, C++ und MFC, Visual C++

Zunächst soll die Entwicklung in der Technik der Windows-Programmierung mit einfachen Beispielen in C bzw. C++ illustriert werden. Ähnliche Entwicklungen über die Objektorientierung zur visuellen Umgebung lassen sich auch in anderen Sprachfamilien nachvollziehen.

8.1.1 Windows-Programmierung in C

Die klassische Programmiersprache für Windows-Programme ist C, ein damit erstelltes Minimal-*HelloWorld* lässt sich – wie das entsprechende C-Einstiegsprogramm unter DOS, UNIX, Linux – auf wenige Zeilen kondensieren:

```
5  #include <windows.h>
```

[44] Die in diesem Skriptum verwendeten Soft- und Hardwarebezeichnungen sind in den meisten Fällen auch eingetragene Warenzeichen und unterliegen als solche den gesetzlichen Bestimmungen.

```
 6  int WINAPI WinMain(HINSTANCE I, HINSTANCE PrevI, LPSTR CL, int CmdShow )
 7  {
 8    MessageBox(NULL, "Hello Windows!", "Quite Simple", MB_OK);
 9    return 0;
10  }
```

Das Hauptprogramm ist immer `WinMain()`, das als `WINAPI` oder `PASCAL` klassifiziert werden muss (Art der Parameterübergabe). Die übergebenen Parameter betreffen Mehrfachaufrufe des Programms (`I` und `PrevI`), Befehlszeile (`CL`) und Sichtbarkeit (`CmdShow`).

Das Ergebnis ist dann auch nicht aufregender als beim Standard-*HelloWorld* unter anderen Systemen, zeigt aber schon, dass man auf relativ einfache Weise vorgefertigte Objekte mit gewisser Basisfunktionalität verwenden kann – so man denn weiß, wie die heißen. Nützlich ist das Minimalprogramm als Test für die richtige Funktion von Compiler, Linker etc.

Soll das Windows-Programm mehr tun, als eine schlichte Meldung von sich zu geben[45], muss man einen größeren Aufwand treiben. `WinMain()` wird deutlich umfangreicher, da einige Eigenschaften des Programms und des zugehörigen Fensters definiert werden müssen.

```
 9  int WINAPI WinMain(HANDLE I, HANDLE IPrev, LPSTR CL, int cmdShow)
10  {
11    MSG msg;
12    WNDCLASS wc;
13    HWND hwnd;
14    if (IPrev == NULL) {
15      memset(&wc, 0, sizeof(wc));
16      wc.lpszClassName = "Hello";
17      wc.hInstance = I;
18      wc.hCursor = LoadCursor(NULL, IDC_ARROW);
19      wc.hIcon = LoadIcon(I, "UNIICON");
20      wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
21      wc.style = 0;
22      wc.lpfnWndProc = MyWndProc;
23      if(!RegisterClass(&wc)) return FALSE;
24      }
25    hwnd = CreateWindow("HELLO", "HELLO", WS_OVERLAPPEDWINDOW,
26                    CW_USEDEFAULT, 0, CW_USEDEFAULT, CW_USEDEFAULT,
27                    NULL, NULL, I, NULL);
28    ShowWindow(hwnd, cmdShow);
29    UpdateWindow(hwnd);
30    while (GetMessage(&msg, NULL, 0, 0))
31      DispatchMessage(&msg);
32    return msg.wParam;
33  }
```

[45] Einfache Dialogfenstern dieser Art lassen sich während der Programmentwicklung vorteilhaft als Debug-Hilfe einsetzen.

Nur einige der hier festgelegten Eigenschaften (Komponenten der `WNDCLASS`-Struktur und Fenstereigenschaften) sind von der jeweiligen Anwendung abhängig, die meisten der obigen Programmzeilen sind in praktisch allen C-Windows-Programmen identisch, insbesondere beispielsweise die letzten Zeilen, die die Fensterdarstellung veranlassen und die eintreffenden Meldungen abarbeiten.

Windows ist ein ereignis- bzw. meldungsorientiertes Betriebssystem, Meldungen (*messages*) lösen Aktionen aus. Daher muss als zentraler 'Briefkasten' für die an das Windows-Programm gerichteten Meldungen eine `CALLBACK`-Funktion bereitgestellt werden, die die eintreffenden Meldungen analysiert und die notwendigen Aktionen veranlasst.

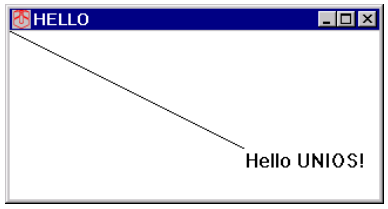
```
46  LRESULT CALLBACK MyWndProc(HWND hwnd, UINT msg, WPARAM wP, LPARAM lP)
47  {
48    switch(msg) {
49      case WM_PAINT:
50        Paint(hwnd);
51        return(0);
52      case WM_DESTROY:
53        PostQuitMessage(0);
54        return(0);
55      default:
56        return(DefWindowProc(hwnd, msg, wP, lP));
57    }
58  }
```

Im vorliegenden Beispiel sind nur die Meldungen `WM_PAINT` und `WM_DESTROY` berücksichtigt, alle anderen werden an die Standardfunktion `DefWindowProc()` weitergeleitet.

Schließlich muss die von `WM_PAINT` veranlasste Aktion `Paint()` noch definiert werden, Einfachst-Grafik und Schrift:

```
35  void Paint(HWND hwnd)
36  {
37    PAINTSTRUCT paintStruct;
38    HDC hDC = BeginPaint(hwnd, &paintStruct);
39    if (hDC != NULL) {
40      LineTo (hDC, 200, 100);
41      TextOut (hDC, 200, 100, "Hello UNIOS!", 12);
42      EndPaint(hwnd, &paintStruct);
43      }
44  }
```

Damit ist das Programm komplett mit seinen wesentlichen Teilen: der Hauptfunktion `WinMain()`, der `CALLBACK`-Funktion mit der Meldungsauswahl und der Funktion `Paint()`, die die gewünschte Arbeit verrichtet. Das Ergebnis ist das nebenstehende Fenster, das nach der Zeichenaktion keine weiteren Ereignisse vorsieht.



8.1.2 Objektorientiert mit C++ und MFC

Über die Bibliothek der *Microsoft Foundation Classes (MFC)* werden einige der immer gleichartigen Grundeigenschaften vordefiniert in einigen Basisklassen bereitgestellt.

Beim Einfachstprogramm kann dies gegenüber C – sieht man einmal von den kürzeren Parameterlisten ab – noch keine wesentliche Verbesserung bringen:

```
2  #include "afxwin.h"
3  class CApp : public CWinApp {
4    virtual BOOL InitInstance() {
5      AfxMessageBox("Hello Win !");
6      return FALSE;
7    };
8  } theApp;
```

Das zweite Beispiel dagegen stellt sich im Vergleich zu C deutlich übersichtlicher dar. Zuerst die Deklaration der benötigten Basisobjekte Anwendung (`theApp`) und zugehöriges Fenster (`FW`):

```
2  #include "afxwin.h"
3
4  class CApp : public CWinApp {
5    virtual BOOL InitInstance();
6  } theApp;
7
8  class CFWin : public CFrameWnd {
9    LRESULT WindowProc( UINT message, WPARAM wParam, LPARAM lParam );
10   void Paint();
11 } FW;
```

Die Initialisierungsfunktion verbindet die Anwendung mit dem Fenster und startet anschließend die Fensterdarstellung mit den gewünschten Eigenschaften.

```
13 BOOL CApp::InitInstance()
14 {
15   m_pMainWnd = &FW;
16   FW.Create(NULL, "MFC Example");
17   FW.ShowWindow(m_nCmdShow);
18   FW.UpdateWindow();
19   return TRUE;
20 }
```

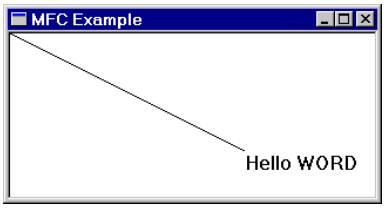
Zur Abarbeitung der Meldungen wird die in der Basisklasse `CFrameWnd` schon vorhandene Funktion `WindowProc()` überladen:

```
22 LRESULT CFWin::WindowProc( UINT message, WPARAM wParam, LPARAM lParam )
23 {
```

```
24    switch (message) {
25      case WM_PAINT:
26        Paint();
27        return 0;
28      case WM_DESTROY:
29        PostQuitMessage(0);
30        return 0;
31      default:
32        return DefWindowProc(message, wParam, lParam);
33      }
34  }
```

Zum Schluss wieder die 'Arbeits'-Funktion `Paint()`. Die Ähnlichkeit mit der korrespondierenden C-Funktion (Seite 125) resultiert in einem entsprechenden Gesamtergebnis.

```
36  void CFWin::Paint()
37  {
38    PAINTSTRUCT p;
39    CDC * dc = BeginPaint(&p);
40    dc->LineTo(200,100);
41    dc->TextOut(200,100,"Hello WORD");
42    EndPaint(&p);
43  }
```



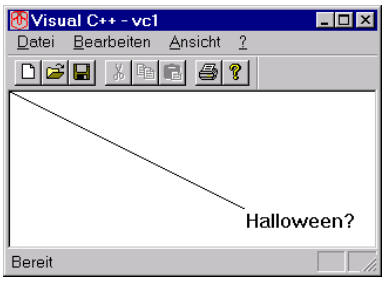
8.1.3 Programmunterstützt in Visual C++

In den bisherigen Programmbeispielen in C und C++ wurde deutlich, dass ein über eine einfache *MessageBox* hinausgehendes Windows-Programm einen fast immer gleichen Routineaufwand erfordert, bis ein lauffähiges Programmgerüst erstellt ist. Dieser Anfangsaufwand kann dadurch vereinfacht werden, dass man das Grundgerüst kopiert[46]; das Erstellen der `switch`-Struktur für die Meldungsabarbeitung und vor allem die Deklaration und Definition der benötigten Funktionen bleibt aber eine aufwendige und vor allem sehr fehleranfällige Arbeit. Da ein Großteil des Arbeitsaufwands formal beschrieben werden kann, liegt es nahe, diesen Teil dem Rechner, d. h. einer darauf spezialisierten Programmierumgebung zu überlassen.

Visual C++ ist eine solche Programmierumgebung, wird sie richtig eingesetzt, können selbst umfangreichere Windows-Programme mit minimalem Programmieraufwand realisiert werden. Wichtig ist dafür allerdings, die vielfältigen Möglichkeiten der Entwicklungsumgebung zu kennen und sie auch zu nutzen, aber auch, sich in erster Näherung mit den Vorgaben der bereitgestellten vorgefertigten Klassen (MFC) zufrieden zu geben.

[46] In klassischen Programmieranleitungen wurde meist ein Basisprogramm (`generic.c` oder ähnlich) bereitgestellt – und meist auch seitenfüllend abgedruckt, das der Anwender als Kopiervorlage benutzen konnte.

Für nebenstehende Anwendung mussten nur die Zeilen 61 und 62 an der richtigen Stelle eingefügt werden.




```
57  void CVc1View::OnDraw(CDC* pDC)
58  {
59    CVc1Doc* pDoc = GetDocument();
60    ASSERT_VALID(pDoc);
61    pDC->LineTo(200,100);
62    pDC->TextOut(200,100,"Halloween?");
63  }
```

Ausgenutzt wurde, dass Visual C++ ein komplettes Ausgangsgerüst für ein Windows-Programm bereitstellt. Die vielseitigen weiteren leistungsfähigen Programmierhilfsmittel der Entwicklungsumgebung sind in der inzwischen sehr umfangreichen Literatur[47] zu Visual C++ ausführlich beschrieben (z. B. in [50]).

8.2 Dialogorientierte Programme

Der MFC-Anwendungsassistent (*MFC AppWizard*) kann fertige Grundgerüste für drei Typen von Programmen erstellen: Einzeldokument-, Mehrfachdokument- und Dialog-basierte. Sie unterscheiden sich in den verwendeten Basisfensterklassen. Für Testzwecke, aber auch für einfachere Anwendungen sind dialogorientierte Programme sehr gut geeignet, da sie für die jeweils gewünschte Funktionalität relativ rasch weitgehend visuell erstellt werden können. An einem einfachen Beispiel (Eingabe, Berechnung, Ausgabe) soll dies erläutert werden.

Nach dem Start der Entwicklungsumgebung mit der Auswahl (*File→New*) eines neuen Projekts des Typs *MFC AppWizard (exe)*, Namensvergabe und Festlegung des Arbeitsverzeichnisses wird im *AppWizard*-Fenster der Punkt ‘Dialog based’ markiert (Abbildung 68) und der Erstellungsdialog zu Ende geführt.

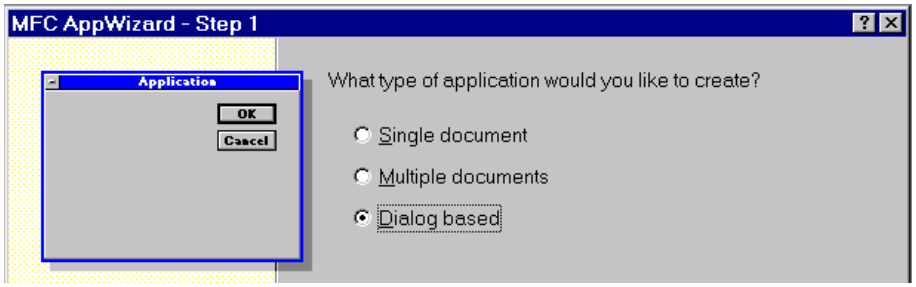


Abbildung 68: Startfenster bei Auswahl eines neuen Projekts des Typs MFC AppWizard (exe), *für Dialog-Programme wird ‘Dialog based’ gewählt.*

[47] Bei der Auswahl eines Buches sollte man darauf achten, ob der Autor sich auf C++ und MFC konzentriert oder aber – was leider auch teilweise vorkommt – zunächst umfänglich C-Programme alten Stils aus Vorgängerbüchern wiederverwertet.

8.2.1 Visuelle Dialogerstellung mit dem Ressourceneditor

Der benötigte Dialog wird visuell mit dem Ressourceneditor erstellt, die Vorgehensweise ist in den folgenden Abbildungen dargestellt:

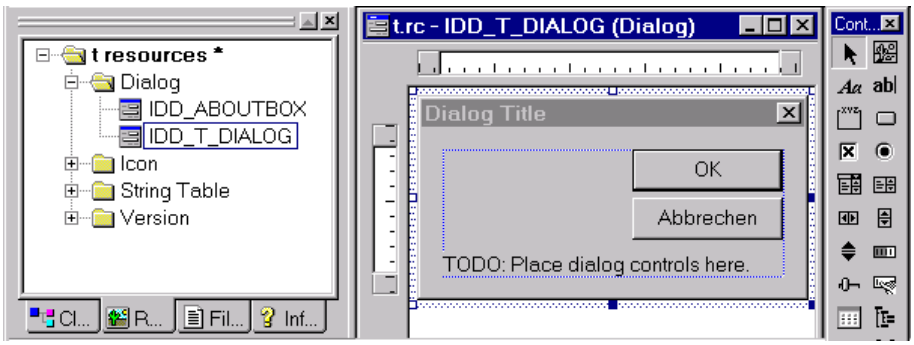


Abbildung 69: Aus der Ressourcenübersicht (links) wird der vorgefertigte Hauptdialog im Ressourceneditor geöffnet und interaktiv erweitert; die wichtigsten Objekte können aus der Werkzeugleiste (rechts) ausgewählt werden.

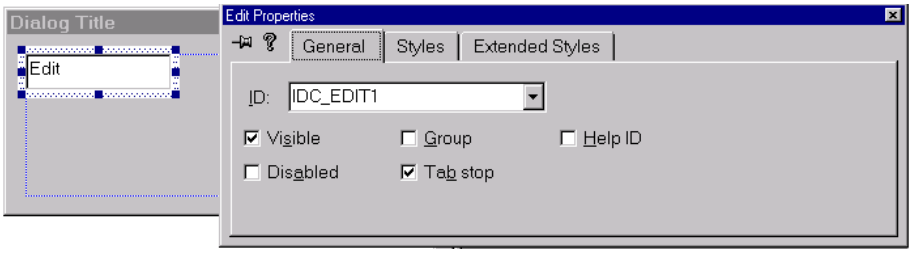


*Abbildung 70: Zunächst wird ein veränderbares Textfeld (*Edit*) eingefügt, und es werden dessen Eigenschaften festgelegt (das obige Eigenschaftsfeld zeigt die Voreinstellungen, beispielsweise kann statt des vom* Wizard *eingestellte Namens* `IDC_EDIT1` *ein sinnvollerer gewählt werden).*

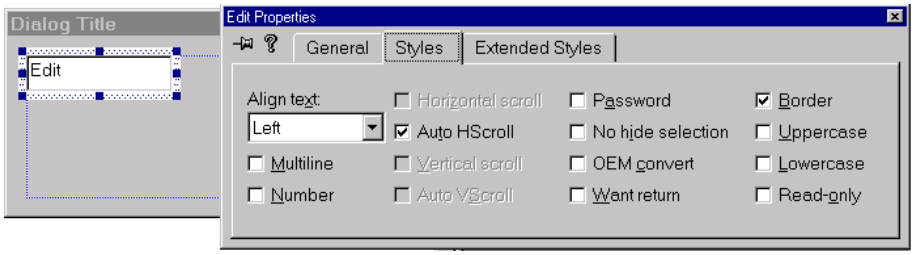


Abbildung 71: Die zweite Eigenschaftsseite: bei Zahlen macht es Sinn, die Textausrichtung von Left *auf* Center *oder* Right *zu ändern.*

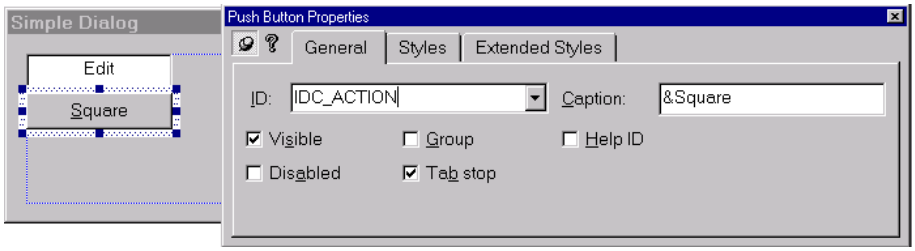


Abbildung 72: Einfügen einer Drucktaste mit der Beschriftung 'Square' und dem Namen `IDC_ACTION`*. Durch '&' wird ein Tastaturkürzel definiert (Alt-S).*

8.2.2 Funktionen und Variablen

Nach der Konstruktion der Benutzeroberfläche werden Funktionen und Variablen an die neuen Objekte gebunden. Dazu wird der Klassen-Assistent (*Class Wizard*) aufgerufen. In seinem Submenü *Member Variables* wird dem erstellten Textfeld mit *Add Variable* die Integervariable `m_Count` zugewiesen (Abbildung 73).

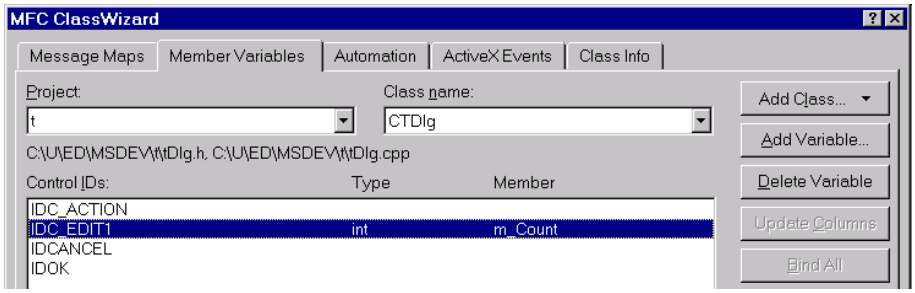


Abbildung 73: Verbinden einer Variablen mit dem Textfeld des Dialogs.

Im Submenü *Message Maps* wird an den Tastendruck auf die *Square*-Taste (`IDC_ACTION`, `BN_CLICKED`) eine Funktion gebunden (*Add Function*). Das *Wizard*-Programm schlägt dafür einen vom Tastennamen `IDC_ACTION` abgeleiteten Funktionsnamen – `OnAction()`

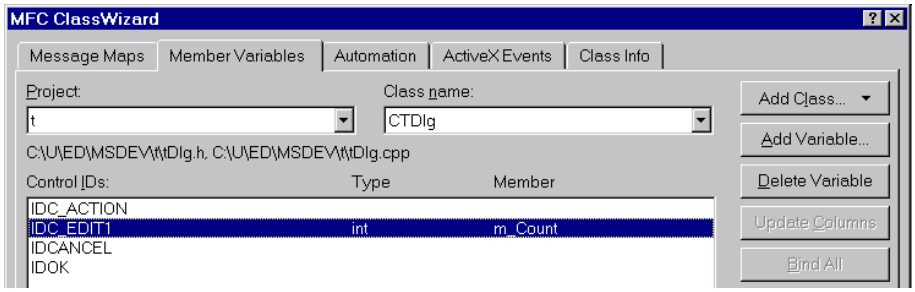


Abbildung 74: Submenü Message Maps *des Klassenassistenten: Mit* Add Function *wird eine Funktion an ein Ereignis gebunden, mit* Edit Code *der Funktionscode erstellt.*

– vor, mit*Edit Code* kommt man dann im Texteditor zum vorgefertigten Funktionsrumpf und kann den Funktionscode implementieren. Im vorliegenden Beispiel sind das die drei Zeilen 175–177 innerhalb der Funktion `OnAction()` (Abbildung 75).

```
173  void CTDlg::OnAction()
174  {
175    UpdateData(true);
176    m_Count *= m_Count;
177    UpdateData(false);
178  }
```

Abbildung 75: Funktionscode und fertige Benutzeroberfläche für das Programmbeispiel.

Wie schon durch die Bezeichnung nahegelegt, wird durch Drücken der *Square*-Taste der Fensterinhalt quadriert. Die Funktion `UpdateData()` synchronisiert die Variablen zwischen Fenster und Programm, die Richtung wird durch `true` (Dialogfenster→Programm, Voreinstellung) bzw. `false` (Programm→Dialogfenster) festgelegt.

8.3 MFC-Zeitfunktionen

Neben den in Abschnitt 7.3 beschriebenen Windows-Funktionen zur Zeitsteuerung stehen in den *Microsoft Foundation Classes* einige vereinfachte Klassen und Methoden zur Verfügung, mit denen zeitliche Abläufe gesteuert werden können.

8.3.1 Systemzeit

Korrespondierend zu den Zeit-Management-Funktionen in C (`time.h`) definieren die *Microsoft Foundation Classes* die beiden Klassen `CTime` und `CTimeSpan` für den objektorientierten Umgang mit absoluten und relativen Zeiten (ausführliche Beschreibung in der Online-Hilfe). Die Funktion

```
void CAnyWnd::OnTimer() {
    CTime now = CTime::GetCurrentTime();
    CString s;
    s.Format ( "%02d:%02d:%02d", now.GetHour(),
               now.GetMinute(), now.GetSecond() );
    SetWindowText ( s );
}
```

etwa würde in der Titelleiste von `AnyWnd` die aktuelle Zeit darstellen. Die für eine regelmäßige Aktualisierung notwendige periodische `WM_TIMER`-Nachricht kann man durch

```
SetTimer ( 1, 1000, NULL );
```

veranlassen (s. nächsten Abschnitt).

8.3.2 Timer

Taktgeber (Timer) können mit etwas vereinfachten Methoden eingerichtet werden, da `SetTimer()` und `KillTimer()` Klassenmethoden der Basisklasse `CWnd` sind. Die Zeitauflösung und damit auch die minimal einstellbare Taktrate beträgt derzeit 10 msec. Nachstehendes Programmfragment verdeutlicht die etwas einfachere Verwendung:

```
static UINT MyTimer = 1;
const UINT Elapse = 100;                        // timeout 100 msec
...
void CALLBACK TiProc (HWND h, UINT msg, UINT timer, DWORD systime)
{ Do what you want to do every 100 msec }
...
MyTimer = SetTimer (MyTimer, Elapse, TiProc); // install timer
...
KillTimer (MyTimer);                            // deinstall timer
```

8.4 Dokumentorientierte Programme

Für einfache Abläufe sind dialogorientierte Programme sehr gut geeignet, vor allem auch rasch zu programmieren. Falls man jedoch innerhalb des Programms Daten graphisch darstellen, Dateien lesen oder schreiben, die Zwischenablage benutzen oder ausdrucken möchte, wird es zweckmäßig, einen anderen vom *MFC AppWizard* bereitgestellten Programmprototypen zu verwenden, dokumentorientierte Programme. Meist reicht es aus, mit einem einzelnen Dokumentfenster zu arbeiten, dazu wählt man im Startfenster des Wizards den Punkt 'Single document' an (Abbildung 76) und führt den Erstellungsdialog mit den vorgeschlagenen Einstellungen zu Ende.

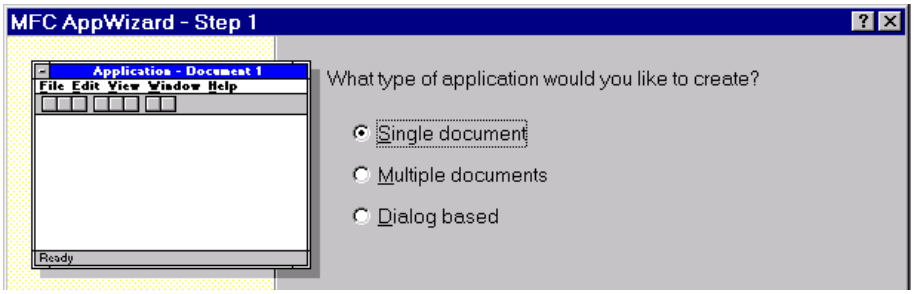


Abbildung 76: Startfenster bei Auswahl eines neuen Projekts des Typs MFC AppWizard (exe), *für dokumentorientierte Programme mit* einem *Dokumentfenster wird 'Single Dokument' gewählt.*

Wie auch bei dialogorientierten Programmen wird eine komplette Programmstruktur erstellt, die schon alle Basiselemente einer typischen Windows-Anwendung im richtigen

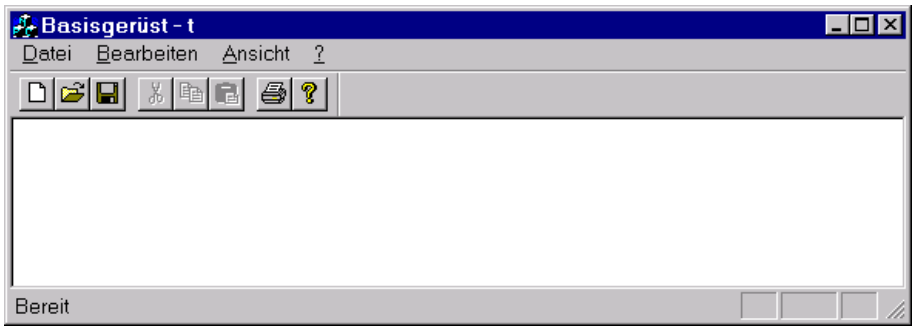


Abbildung 77: Vom AppWizard *zusammengestellte Basiselemente einer 'Single-Document'-Anwendung.*

'Look and Feel' enthält (Abbildung 77). So gibt es unter anderem eine vorgefertigte Menüleiste mit den wichtigsten Menüpunkten, eine Werkzeugleiste mit einigen Drucktasten und eine unten im Fenster eingeblendete Statuszeile. Weiterhin ist dafür gesorgt, dass der Name des aktuellen Dokuments zusammen mit dem Programmnamen im Fenstertitel erscheint. Ergänzungen in der Benutzeroberfläche und Erweiterungen in den vorgefertigten Prototypfunktionen machen aus der Grundstruktur eine Anwendung mit der gewünschten Funktionalität.

Die Programmfunktionen sind bei dokumentorientierten Programmen auf mehrere Basisobjekte verteilt: Die Anwendung (Basisklasse `CWinApp`), das Hauptfenster (Basisklasse `CFrameWnd`), das eigentliche Dokument (Basisklasse `CDocument`) und die Dokumentdarstellung (Basisklasse `CView`). Ergänzungen werden zweckmäßigerweise dort implementiert, wo sie logisch hingehören. Insbesondere sollte man gut zwischen Dokument und Darstellung abgrenzen (Datei-Management in der Dokument-Klasse, Drucken wie auch Bildschirmdarstellung in der View-Klasse).

8.4.1 Dateizugriff

Alles, was man dazu braucht, ist vorgefertigt vorhanden, wenn man das Standard-Dateimenü verwendet. Dokumente werden durch die Methode `Serialize()` in einem Archiv (Typ `CArchive`) abgelegt oder daraus geladen, das seinerseits mit einem Dateiobjekt (Typ `CFile`) verbunden ist. Dateiname und -verzeichnis werden über den gewohnten Standarddialog festgelegt.

`Serialize()` ist innerhalb der Dokumentklasse als leeres Gerüst vorgefertigt:

```
void CAnyDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring()) {
    // TODO: add storing code here
  }
```

```
    else {
      // TODO: add loading code here
    }
  }  .
```

Eine Implementierung, die `nPoints` Integer-Datenwerte `Y[i]` aus dem Objekt `Osc` zeilenweise in Textform speichert oder liest, könnte so aussehen:

```
void CAnyDoc::Serialize(CArchive& ar)
{
  CString s;
  if (ar.IsStoring()) {
    for (int i=0; i<Osc.nPoints; i++) {
      s.Format("%3d\r\n", Osc.Y[i]);
      ar.WriteString (s);
    }
  }
  else {
    for (int i=0; ar.ReadString(s) && (i<Osc.MaxPoints()); i++)
      Osc.Y[i] = atoi(s);
    Osc.nPoints = i;
  }
}  .
```

Neben den Methoden `WriteString()` und `ReadString()` bietet `CArchive` noch `Write()` und `Read()` für binären Zugriff sowie die Streamoperatoren `<<` und `>>` für die gängigsten Variablentypen.

8.4.2 Graphik

Die graphische Darstellung der gemessenen Daten ist ein essentieller Teil bei jedem physikalischen Experiment. Die für Veröffentlichungen, Diplom- oder Doktorarbeiten benötigten Präsentationsgraphiken können sehr gut mit darauf spezialisierten Programmen aus dem Public-Domain-Bereich wie GnuPlot oder kommerziellen Programmen wie Excel, Origin, EasyPlot, MATLAB erstellt werden. Daneben sind jedoch – um eine physikalische Messung zu überwachen und zu dokumentieren – einfache Bildschirmgraphiken notwendig, die das Messprogramm zeitgleich mit der Messung direkt aus den Rohdaten liefern sollte. Solche Einfachgraphiken lassen sich ohne großen Aufwand mit den Windows-Hilfsmitteln realisieren.

Graphische Darstellungen in einer Anwendung sollten sich ‘windowskonform’ verhalten, d. h. beispielsweise nach einer Änderung der Fenstergröße sofort in der veränderten Größe neu gezeichnet oder nach einer Fensterüberdeckung umgehend regeneriert werden. Solche Aktualisierungen werden im betroffenen Fenster jeweils durch eine `WM_PAINT`-Nachricht ausgelöst, die vom System an das Fenster geschickt wird. In der `CView`-Klasse wird dadurch

letztlich die Methode `OnDraw()` angestoßen, die dafür zuständig ist, den Fensterinhalt zu zeichnen. Für die Programmierung heißt das andererseits, dass alle Graphikaktionen mit permanentem Ergebnis ausschließlich über `OnDraw()` ausgeführt werden sollten. Eine Methode, die für die Datenerfassung zuständig ist, muss sich daher darauf beschränken, die Daten zu sammeln und bereitzustellen sowie von Zeit zu Zeit die Aktualisierung der graphischen Darstellung zu veranlassen, beispielsweise durch Aufruf der Fensterfunktion `Invalidate()`.

Gerätekontext: Fast alle Informationen und viele Werkzeuge, die zur graphischen Darstellung notwendig sind, werden von der Gerätekontextklasse `CDC` oder davon abgeleiteten Klassen bereitgestellt. Der jeweils aktuelle Gerätekontext wird beim Aufruf von `OnDraw()` als Parameter übergeben und kann dann innerhalb der Zeichenroutine benutzt werden.

Koordinaten: Gezeichnet wird auf den Bildschirm, genauer in das Client-Fenster. Dessen relative Pixel-Koordinaten liefert die Fenster-Funktion `GetClientRect()`, die sie in einer Struktur des Typs `CRect` ablegt. Man kann direkt in diesem Koordinatensystem arbeiten, indem man die zu zeichnenden Daten in diese Koordinaten umrechnet. Einfacher ist es, die Umrechnung Windows zu überlassen und die Koordinatentransformation indirekt zu spezifizieren. Dazu stellt die `CDC`-Klasse verschiedene Funktionen bereit, die die Abbildung von Benutzer-Koordinaten (*Window*) auf Gerätekoordinaten (*Viewport*) regeln. Die Programmierung kann meist entsprechend dem folgenden Beispiel ablaufen:

```
1  void CAnyView::OnDraw(CDC* pDC) {
2    CRect R;
3    GetClientRect(R);
4    R.DeflateRect(10, 10);
5    pDC->SetMapMode(MM_ANISOTROPIC);
6    pDC->SetWindowExt(Osc.Xmax()-Osc.Xmin(), Osc.Ymin()-Osc.Ymax());
7    pDC->SetWindowOrg(Osc.Xmin(), Osc.Ymax());
8    pDC->SetViewportExt(R.Size());
9    pDC->OffsetViewportOrg(R.left, R.top);
10   ...
```

Zunächst werden die Koordinaten des Client-Fensters festgestellt, Zeile 4 verkleinert die Zeichenfläche um einen 10 Pixel breiten Rand. In Zeile 5 wird die richtige Abbildungsart eingestellt, Zeilen 6 und 7 definieren die Benutzerkoordinaten, 8 und 9 die Gerätekoordinaten. In `SetWindowOrg()` ist berücksichtigt, dass der Ursprung der Gerätekoordinaten oben links ist.

Zeichenfunktionen: Für das Zeichnen von Linien sind die beiden Methoden `MoveTo()` und `LineTo()` zuständig, ein rechteckiger Rahmen könnte mit

```
9    ...
10   pDC->MoveTo(Osc.Xmin(), Osc.Ymin());
11   pDC->LineTo(Osc.Xmax(), Osc.Ymin());
12   pDC->LineTo(Osc.Xmax(), Osc.Ymax());
13   pDC->LineTo(Osc.Xmin(), Osc.Ymax());
```

```
14    pDC->LineTo(Osc.Xmin(), Osc.Ymin());
```

die Daten schließlich mit

```
15    pDC->MoveTo(Osc.X[0], Osc.Y[0]);
16    for (int i=1; i<Osc.nPoints; i++)
17      pDC->LineTo(Osc.X[i], Osc.Y[i]);
```

gezeichnet werden.

Zeichenwerkzeuge: Für farbige Linien braucht man einen farbigen Zeichenstift (*Pen*), er wird als `CPen`-Objekt kreiert und vor Zeichenbeginn selektiert:

```
CPen * RedPen = new CPen ( PS_SOLID, 0, RGB (255, 0, 0) );
pDC->SelectObject ( RedPen );
```

in diesem Fall ein durchgezogener Linientyp der Strichstärke 1 Pixel in Rot.

8.4.3 Drucken

Das Ausdrucken einer Graphik erfolgt ebenfalls über die `OnDraw()`-Methode, allerdings in einem anderen Gerätekontext und vor allem auf ein Gerät mit anderen Koordinaten. Ob gedruckt wird, kann mit der Funktion `IsPrinting()` aus der `CDC`-Klasse nachgefragt werden. Abhängig von der Antwort können Teile der Graphik je nach Gerät unterschiedlich behandelt werden, z. B. könnte eine auf dem Bildschirm farbige Linie auf den Drucker schwarz und strichpunktiert ausgegeben werden. Wichtiger ist es, die Koordinatentransformation geeignet zu ändern, um die Graphik sinnvoll im Ausdruck zu plazieren. Dazu muss man sich die Druckereigenschaften beschaffen, am einfachsten aus dem in `OnPreparePrinting()` übergebenen Druckerinformationsblock, dessen Adresse man sich kopiert:

```
static CPrintInfo * PrintInfo;
...
BOOL CAnyView::OnPreparePrinting(CPrintInfo* pInfo) {
  PrintInfo = pInfo;
  ...
}  ,
```

um daraus innerhalb der `OnDraw()`-Methode die Druckerkoordinaten zu extrahieren:

```
if (pDC->IsPrinting()) {
  R = PrintInfo->m_rectDraw;
  R.DeflateRect(R.Width()/8, R.Height()/8, R.Width()/8, R.Height()/2);
}  .
```

Mit `DeflateRect()` wird hier ein linker, oberer und rechter Rand von etwa 25 mm eingestellt, ein unterer Rand von halber Seitenlänge.

Was sonst noch zum Drucken nötig ist, kann man dem Druckprozessor von Windows überlassen.

8.4.4 Zwischenablage

Ähnlich einfach ist der Transport einer Graphik in die Zwischenablage, um sie in einem anderen Windows-Programm zu verwenden (WinWord, Datenbank etc.).

Zunächst wird der Menü-Unterpunkt 'Edit→Copy' ('Bearbeiten→Kopieren') mit einer Funktion versehen, das erledigt der *Class Wizard* (Abbildung 78).

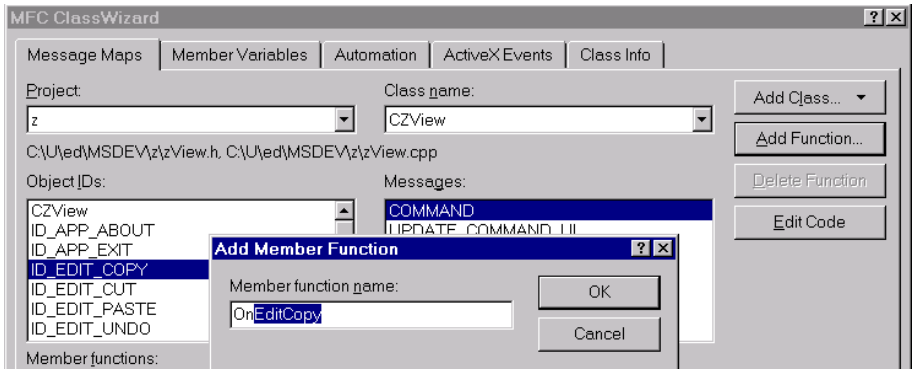


Abbildung 78: Zuweisung einer Funktion zu `ID_EDIT_COPY` *im* Class Wizard.

Dann ist noch etwas Funktionscode nötig:

```
void CAnyView::OnEditCopy() {
  CMetaFileDC p;
  if ( !p.CreateEnhanced( NULL, NULL, NULL, NULL ) )
    return;
  OnDraw (&p);
  HENHMETAFILE hMF = p.CloseEnhanced();
  OpenClipboard();
  EmptyClipboard();
  SetClipboardData(CF_ENHMETAFILE, hMF);
  CloseClipboard();
}  .
```

Es wird ein Metafile-Gerätekontext kreiert und in diesem die Graphik – wie üblich mit `OnDraw()` – erstellt. Der darauf verweisende Zeiger (Handle) `hMF` wird schließlich an die Zwischenablage übergeben.

8.4.5 Dialogfenster

Um die Parameter für ein Messprogramm einzustellen oder um einzelne Daten darzustellen, ist es auch in einem primär dokumentorientierten Programm sinnvoll, mit Dialogfenstern zu arbeiten, da diese sehr flexibel erstellt werden können.

Man muss zwei Typen von Dialogfenstern unterscheiden: *Modale* Dialoge beanspruchen für sich die volle Aufmerksamkeit, eine Rückkehr zum Hauptprogramm ist erst am Ende des Dialogs möglich. Man sollte sie dort einsetzen, wo eine Benutzereingabe auf jeden Fall zu Ende geführt werden muss, bevor der normale Programmablauf fortgesetzt werden kann. *Nichtmodale* Dialoge dagegen können wie zusätzliche parallel arbeitende Programmfenster verwendet werden.

Erstellt werden beide Dialogtypen zunächst graphisch im Ressourceneditor. Mit dem *Class Wizard* wird dann eine zugehörige Klasse (hier: `CParam`, Parameterdialog) eingerichtet, die im allgemeinen Fall von `CDialog` abgeleitet wird, in Spezialfällen aber auch auf anderen Dialog-Klassen basieren kann. Die Klasseneigenschaften werden über die Headerdatei importiert, ein Menüpunkt zum Aufrufen des Dialogs und eine zugehörige Funktion (`OnParam()`) werden eingerichtet.

Bei *modalen* Dialogen ist die Verwendung sehr einfach, ein Dialog-Objekt wird innerhalb der Menüfunktion angelegt und modal aufgerufen:

```
void CAnyView::OnParam() {
  CParam P;
  P.DoModal();
}  .
```

Die Methode `DoModal()` sorgt dafür, dass das Dialogfenster erstellt und dargestellt wird und dass die Parameter (*Member Variables*) bei 'OK' aktualisiert, bei 'Abbrechen' dagegen verworfen werden.

Die Farbe eines Zeichenstiftes könnte in einem *modal* geführten Dialog der abgeleiteten Klasse `CColorDialog` etwa durch die Zeilen

```
static COLORREF COLOR;
...
void CAnyView::OnParam() {
  CColorDialog C;
  if ( C.DoModal()==IDOK )
    COLOR = C.GetColor();
}
...
static CPen * ColorPen = new CPen ( PS_SOLID, 0, COLOR );
```

eingestellt werden.

Bei *nichtmodalen* Dialogen ist dagegen etwas mehr Aufwand nötig. Im einfachsten Fall wird eine zusätzliche `Create()`-Methode implementiert:

```
BOOL CParam::Create(CWnd* pParentWnd) {
  return CDialog::Create(IDD, pParentWnd);
}  ,
```

mit der das global im Programm angelegte Dialog-Objekt an einer zentralen Stelle kreiert wird, beispielsweise im Konstruktor des `View`-Objekts:

```
static CParam P;
CAnyView::CAnyView() {
  P.Create (this);
}  .
```

Mit der folgenden Implementierung von `OnParam()` kann dann die Sichtbarkeit des Dialogfensters geswitcht werden:

```
void CAnyView::OnParam() {
  if (!P.ShowWindow(SW_HIDE))
    P.ShowWindow(SW_SHOWNORMAL);
}  .
```

Da das Dialog-Objekt dauerhaft vorhanden ist – sichtbar oder unsichtbar – muss der Datenabgleich (`P.UpdateData()`) zur richtigen Zeit vom Programm explizit veranlasst werden. Eine geeignete Stelle dafür ist z. B. der Zeitpunkt, an dem ein Messablauf gestartet wird.

Kleine nichtmodale Dialogfenster kann man als 'System modal' (Properties→More Styles) einrichten, dann werden sie nicht durch andere 'normale' Fenster überdeckt.

8.4.6 Benutzeroberfläche

Die Hauptelemente der Benutzeroberfläche eines dokumentorientierten Programms sind in ihrer Grundstruktur vorhanden – Menü, Werkzeugleiste und Statuszeile – und brauchen nur noch geeignet erweitert zu werden. Primäres Bedienelement ist die Menüleiste, alle Erweiterungen sollten zunächst dort als neue Menüpunkte im Haupt- oder einem Submenü implementiert werden. Die Menüfunktionen werden mit dem *Class Wizard* hinzugefügt. Die Werkzeugleiste wird ebenfalls graphisch um neue Tasten erweitert, die üblicherweise mit schon vorhandenen IDs belegt werden und dann Menüpunkte duplizieren, aber auch neue Funktionen realisieren können. Die Einträge in der Prompt-Zeile des 'Property'-Dialogs definieren den Inhalt der Statuszeile zum jeweiligen Menüpunkt (erste Zeile, vor \n), bei Drucktasten der Werkzeugleiste zusätzlich den Erläuterungstext (zweite Zeile, nach \n).

8.4.7 Mehrere Dokumentfenster

Meist reicht für Datenerfassungsprogramme der beschriebene Programmtyp 'Single document' aus. Sobald man aber z. B. mehrere Messdatensätze gleichzeitig darstellen und getrennt voneinander abspeichern will, wird es zweckmäßig, den etwas komplexeren Prototyp 'Multiple document' einzusetzen. Das wird mit einer geänderten Anfangsauswahl im Startfenster des *AppWizard* veranlasst.

Neben den schon bekannten Basisobjekten Anwendung (Basisklasse `CWinApp`), Hauptfenster (Basisklasse `CMDIFrameWnd`), Dokument (Basisklasse `CDocument`) und Dokumentdarstellung (Basisklasse `CView`) wird dann ein Subfenster-Objekt (Basisklasse `CMDIChildWnd`)

deklariert, das nun Dokument und View enthält und logisch zwischen Hauptfenster und Dokument bzw. View eingeschoben ist . Während beim Einzeldokumentprogramm alle Objekte nur einfach vorhanden sind, wird beim Mehrfachdokumentprogramm jeweils ein neuer Objektsatz Subfenster, Dokument und View erstellt, wenn ein neues Dokument (z. B. mit 'Datei→Neu') angelegt wird.

Das muss bei der Implementierung der Erweiterungen berücksichtigt werden: Sehr viel strenger als beim Einzeldokumentprogramm ist zwischen den verschiedenen Objekten zu unterscheiden, insbesondere zwischen solchen, die einfach, und solchen, die mehrfach vorhanden sind. Objekte zur Bedienung des Experiments dürfen nur einfach angelegt werden, sie sollten also z. B. im Hauptfenster implementiert werden. Gemessene Datensätze und zugehörige Parameter dagegen liegen meist mehrfach vor, werden daher im Rahmen des Dokumentobjekts realisiert.

Zur Verbindung zwischen den gerade aktuellen Objekten enthalten die einzelnen Klassen verschiedene Methoden, die meist einen Zeiger auf das betreffende Objekt zurückgeben:

`CMDIFrameWnd::GetActiveFrame()` zeigt auf das augenblicklich aktive Subfenster,

`CMDIChildWnd::GetActiveDocument()` auf das im Subfenster aktuelle Dokument und

`CMDIChildWnd::GetActiveView()` auf das dort gerade aktive View-Objekt.

`CView::GetDocument()` vermittelt die Verbindung des View-Objekts zum dargestellten Dokument.

`CDocument::GetFirstViewPosition()` und `CDocument::GetNextView()` liefern die verschiedenen Darstellungen des Dokuments (ein Dokument kann mit mehreren Views verbunden sein).

`CDocument::UpdateAllViews(NULL)` aktualisiert alle mit dem Dokument verbundenen Views und sollte nach Änderungen im Dokument bzw. in den Daten aufgerufen werden.

Die ersten drei sind Methoden der Basisklasse `CFrameWnd`, mithin können alle obigen Funktionen auch in Einzeldokumentprogrammen verwendet werden.

8.4.8 Bitmap-Graphiken

Einfache funktionale Zusammenhänge (wenige Messgrößen in Abhängigkeit von einem eindimensionalen experimentellen Parameter), wie sie in Experimenten meist vorliegen, lassen sich mit den in Kapitel 8.4.2 dargelegten Möglichkeiten der Linien-Graphiken zufriedenstellend skizzieren. Bei mehrdimensionalen Abhängigkeiten, z. B. Messungen mit Array-Detektoren, sind dagegen flächenhafte Darstellungen sehr viel anschaulicher. Komplexe Präsentationsgraphiken sollte man auch hier kompetenten Programmen wie MATLAB, AVS-Express, Iris-Explorer überlassen, insbesondere wenn höherdimensionale Zu-

sammenhänge durch geeignete Hyperflächen veranschaulicht werden sollen. Einfache Bitmap-Graphiken können aber mit vertretbarem Aufwand in Messprogrammen realisiert werden, wenn dies zur Verfolgung der Messung oder zur Vorauswahl von Messergebnissen nötig ist.

Device Independent Bitmap (DIB): Obwohl die *Microsoft Foundation Classes* der Typ `CBitmap` zur Verfügung stellen, ist es einfacher, mit geräteunabhängigen Bitmaps (DIBs) zu arbeiten. Windows stellt dafür einige Strukturen und Funktionen bereit. DIBs sind insofern geräteunabhängig, als sie alle wichtigen Informationen zur Darstellung enthalten, z. B. auch die Farbtabelle und implizit die gewünschte reale Bildgröße. DIBs werden durch eine Struktur vom Typ `BITMAPINFOHEADER`, eine Farbtabelle und das Datenfeld des eigentlichen Bildinhalts beschrieben. Relativ komplex ist der `BITMAPINFOHEADER` aufgebaut:

```
typedef struct tagBITMAPINFOHEADER{ // bmih
   DWORD  biSize;
   LONG   biWidth;
   LONG   biHeight;
   WORD   biPlanes;
   WORD   biBitCount;
   DWORD  biCompression;
   DWORD  biSizeImage;
   LONG   biXPelsPerMeter;
   LONG   biYPelsPerMeter;
   DWORD  biClrUsed;
   DWORD  biClrImportant;
} BITMAPINFOHEADER;  .
```

`biSize` enthält die Größe der Struktur (`sizeof(BITMAPINFOHEADER)`), `biWidth` die Breite und `biHeight` die Höhe der Bitmap (in Pixel), `biPlanes` die Anzahl der Farbebenen (=1), `biBitCount` die Zahl der Bits pro Punkt (8 bei 256 Farben), `biCompression` den Kompressionsmodus (`BI_RGB` für unkomprimiert), die nächsten drei kann man auf null setzen, `biClrUsed` und `biClrImportant` geben an, wieviele Farben benutzt und wieviele davon wichtig sind.
Die Farbtabelle ist ein Feld aus `RGBQUAD`s

```
typedef struct tagRGBQUAD { // rgbq
  BYTE    rgbBlue;
  BYTE    rgbGreen;
  BYTE    rgbRed;
  BYTE    rgbReserved;
} RGBQUAD;  ,
```

das mit dem `BITMAPINFOHEADER` zur `BITMAPINFO` zusammengefasst ist:

```
typedef struct tagBITMAPINFO { // bmi
   BITMAPINFOHEADER bmiHeader;
```

```
    RGBQUAD          bmiColors[1];
} BITMAPINFO;  .
```

Der Bildinhalt wird durch ein zweidimensionales Datenfeld beschrieben, für eine quadratische 256-Farben-Bitmap mit der Seitenlänge `BSIZE` z. B. durch `BYTE b[BSIZE][BSIZE]`. Dargestellt auf einem Ausgabegerät wird die DIB dann mit der Funktion `StretchDIBits()`, die die Bitmap auch gleich auf die gewünschte Größe bringt:

```
int StretchDIBits (
  HDC hdc,          // handle of device context
  int XDest,        // x-coordinate of upper-left corner of dest. rect.
  int YDest,        // y-coordinate of upper-left corner of dest. rect.
  int nDestWidth,   // width of destination rectangle
  int nDestHeight,  // height of destination rectangle
  int XSrc,         // x-coordinate of upper-left corner of source rect.
  int YSrc,         // y-coordinate of upper-left corner of source rect.
  int nSrcWidth,    // width of source rectangle
  int nSrcHeight,   // height of source rectangle
  CONST VOID *lpBits,             // address of bitmap bits
  CONST BITMAPINFO *lpBitsInfo,   // address of bitmap data
  UINT iUsage,      // usage
  DWORD dwRop       // raster operation code
);  .
```

Farbtabelle: Eine geeignete Farbtabelle lässt sich mit einer `for`-Schleife erstellen. Eine Graustufentabelle wird mit

```
for (int n=0; n<256; n++) {
  bmi.c[n].rgbBlue = n;
  bmi.c[n].rgbGreen = n;
  bmi.c[n].rgbRed = n;
  bmi.c[n].rgbReserved = 0;
};  ,
```

eine mit kupfergetöntem Intensitätsverlauf durch

```
for (int n=0; n<256; n++) {
  bmi.c[n].rgbBlue = n/2;
  bmi.c[n].rgbGreen = n*2/3;
  bmi.c[n].rgbRed = n;
  bmi.c[n].rgbReserved = 0;
};
```

realisiert. Zur Erhöhung des Farbkontrastes kann man Falschfarbendarstellungen verwenden, so wird mit

```
for (int n=0; n<256; n++) {
```

```
    bmi.c[n].rgbBlue = 255-n;
    bmi.c[n].rgbGreen = n;
    bmi.c[n].rgbRed = 255-n;
    bmi.c[n].rgbReserved = 0;
  };
```

eine Farbtabelle erzeugt, die von Magenta nach Grün wechselt, mit

```
  int dB = 8, dG = 0, dR = 0;
  int blue = 0, green = 0, red = 0;
  for ( int n=0; n<256; n++ ) {
    switch ( n ) {
    case  32:  dB =  0;  dG =  8;           break;
    case  64:  dB = -8;  dG =  0;           break;
    case  96:  dB =  0;           dR = 8;   break;
    case 128:           dG = -4;  dR = 0;   break;
    case 192:  dB =  8;  dG =  0;           break;
    case 224:  dB =  0;  dG =  8;           break;
    }
    blue += dB;  green += dG;  red += dR;
    bmi.c[n].rgbBlue = (blue>255) ? 255 : (blue<0) ? 0 : blue;
    bmi.c[n].rgbGreen = (green>255) ? 255 : (green<0) ? 0 : green;
    bmi.c[n].rgbRed = (red>255) ? 255 : (red<0) ? 0 : red;
  }
```

eine Tabelle, die sehr kontrastreich über alle Farben läuft (vgl. dazu auch [51]).

Beispiel: In einem 'Multiple-Document'-Programm soll ein quadratisches Intensitätsfeld dargestellt werden. Dazu werden in der Dokument-Klasse die nötigen Datenstrukturen angelegt:

```
public:
  BYTE b[BSIZE][BSIZE];
  struct {
    BITMAPINFOHEADER bmih;
    RGBQUAD c[256];
  } bmi;
```

ein Feld für die Daten und eine Struktur für die Bitmapinfo. Der `BITMAPINFOHEADER` wird im Konstruktor des Dokuments besetzt:

```
CMulDoc::CMulDoc() {
  BITMAPINFOHEADER bmih =
    { 0, BSIZE, BSIZE, 1, 8, BI_RGB, 0, 0, 0, 256, 256 };
  memcpy(&bmi.bmih, &bmih, sizeof(BITMAPINFOHEADER));
  bmi.bmih.biSize = sizeof(BITMAPINFOHEADER);
}  ,
```

die Farbtabelle durch eine der Schleifen des vorherigen Abschnitts. Die Bilderstellung erfolgt wie üblich innerhalb der `OnDraw()`-Methode der View-Klasse:

```
StretchDIBits(pDC->m_hDC, xoff, yoff,
  xsize, ysize, 0, 0, BSIZE, BSIZE, GetDocument()->b,
  (BITMAPINFO*) &GetDocument()->bmi, DIB_RGB_COLORS, SRCCOPY);  .
```

`xoff, yoff, xsize, ysize` werden geräteabhängig festgelegt (`pDC->IsPrinting()`). Die Daten werden im Beispielprogramm synthetisch innerhalb des Dokument-Objekts generiert (eine Funktion des Typs $z = \sin(x) \cdot \sin(y)$). Bei einem Messprogramm würde man die Datenerfassung im Hauptfenster starten und die Daten im Dokument-Objekt ablegen, die Aktualisierung der Datendarstellung (während oder nach einer Messung) durch die Funktion `UpdateAllViews(NULL)` aus der Klasse `CDocument` veranlassen. Abbildung 79 zeigt das ideale 'Messergebnis' in drei Subfenstern mit unterschiedlichen Farbtabellen. Bei idealen Daten wird mit jedem der gewählten Farbverläufe ein guter Bildeindruck erzielt.

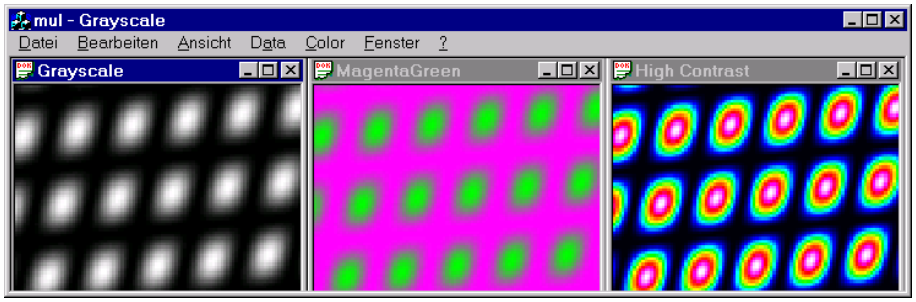


Abbildung 79: Darstellung eines 'idealen' Messergebnisses (Dynamik 8 Bit) in drei unterschiedlichen Farbtabellen. Links: lineare Graustufentabelle, Mitte: Falschfarbenübergang Magenta – Grün, rechts: extremer Farbkontrast.

Anders bei nichtidealen Daten: Der Vorteil einer kontrastreichen Falschfarbendarstellung zeigt sich insbesondere dann, wenn die dargestellte Bitmap eine geringe lokale Dynamik aufweist, diesen Fall demonstriert Abbildung 80.

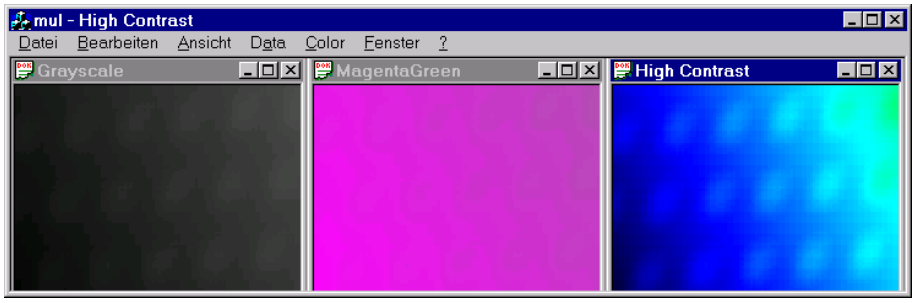


Abbildung 80: Dasselbe Messergebnis mit sehr geringer lokaler Dynamik, nur bei der Darstellung mit überhöhtem Kontrast (rechts) lassen sich Strukturen deutlich erkennen.

Dass aber eine kontrasterhöhende Darstellung nicht immer das Mittel der Wahl ist, zeigt sich bei stark verrauschten Intensitätsfeldern (Abbildung 81).

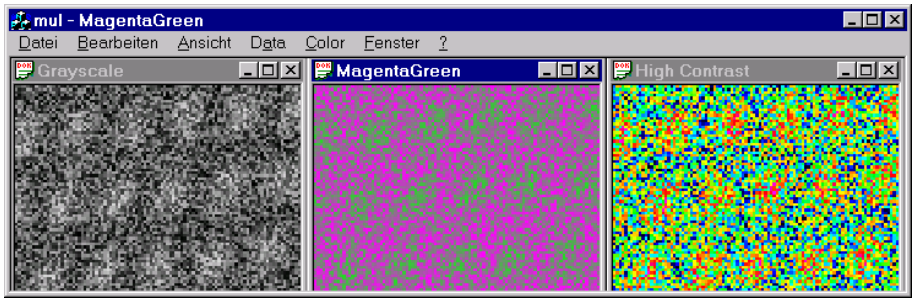


Abbildung 81: Stark verrauschtes Messergebnis: die kontrasterhöhende Darstellung lässt keine Konturen mehr erkennen, vorzuziehen ist eine der beiden kontrastärmeren Darstellungsarten.

BMP-Dateien: Will man Bitmaps abspeichern, macht man das am einfachsten im windowsspezifischen BMP-Format, da dieses Dateiformat exakt dem DIB-Format entspricht. Nach einem kurzen Header, der die Kennung 'BM' und den Offset vom Dateianfang zu den Daten, oft auch die Dateilänge enthält, werden `BITMAPINFOHEADER`, Farbtabelle und Daten hintereinander abgelegt:

```
void CMulDoc::Serialize(CArchive& ar) {
  if (ar.IsStoring()) {
    const char bmptag[] = "BM";
    struct { long filesize, unused, offset; } bmphead;
    bmphead.offset = strlen(bmptag) + sizeof(bmphead) + sizeof(bmi);
    bmphead.unused = 0;
    bmphead.filesize = bmphead.offset + sizeof(b);
    ar.Write(bmptag, strlen(bmptag));
    ar.Write(&bmphead, sizeof(bmphead));
    ar.Write(&bmi, sizeof(bmi));
    ar.Write(&b, sizeof(b));
  }
}  .
```

8.4.9 Fenstereigenschaften

Die Eigenschaften der zu einem Programm gehörenden Fenster werden von der Entwicklungsumgebung einigermaßen sinnvoll festgelegt. Falls man von diesen Voreinstellungen abweichen will, z. B. für die Subfenster ein bestimmte, an die Daten angepasste Größe einstellen will, kann man das am besten innerhalb der Funktion `PreCreateWindow(CREATESTRUCT& cs)` tun. Diese Funktion steht im Hauptfenster und in der Subfensterklasse für Benutzeranpassungen zur Verfügung. Sie führt dem Anwender die Struktur der Fensterei-

genschaften vor, bevor das Fenster real kreiert wird. Beispielsweise kann über die Komponenten `cs.cx` und `cs.cy` die Fenstergröße, über `cs.x` und `cs.y` die Fensterposition eingestellt werden.

8.5 Sockets mit MFC-Unterstützung

Visual C++ hat in den Microsoft Foundation Classes für den direkten Zugriff auf die Socket-Schnittstelle die Basisklasse `CAsyncSocket` und die davon abgeleitete komfortablere Klasse `CSocket` implementiert. Datentransfers laufen in aller Regel über synchrone TCP-Verbindungen ab, daher kann man sich auf die `CSocket`-Klasse beschränken, die geeignete Synchronisations- bzw. Blockungsmechanismen impliziert.

Bei der *AppWizard*-gesteuerten Programmerstellung versichert man sich der *Windows-Open-Services-Architecture*-Unterstützung über Sockets durch die Auswahl des entsprechenden Menüpunkts (Abbildung 82.

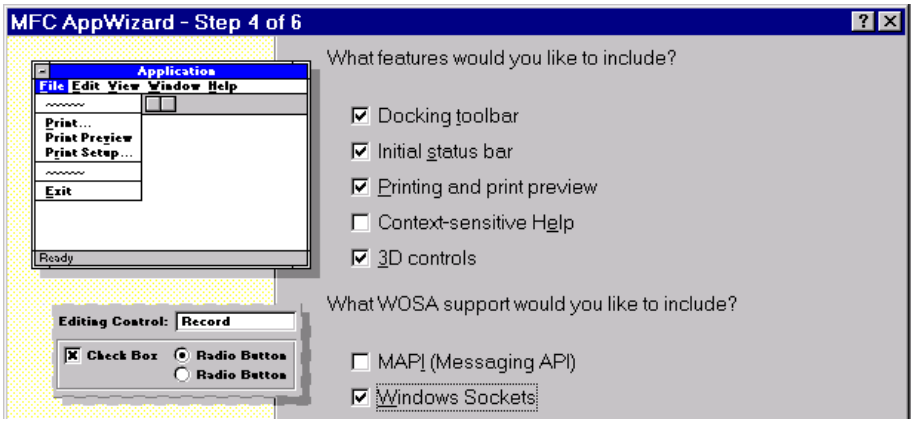


Abbildung 82: Auswahlfenster des MFC AppWizard*: Socket-Unterstützung.*

Damit wird eine Initialisierung des Socket-Systems ins Programm integriert und man kann die beiden Socket-Klassen ohne weiteren Aufwand verwenden.

8.5.1 Server-Socket

Ein Server-Socket sollte ständig empfangsbereit sein, wird daher zweckmäßigerweise als statisches Objekt im Programm angelegt. Einige Methoden von `CSocket` sind zu verändern, man implementiert folglich im *Class Wizard* eine neue Klasse z. B. `CRcv`, die von `CSocket` abgeleitet wird. Ein `CRcv`-Objekt wird im Hauptprogramm statisch angelegt

```
CRcv theReceiver;
```

und im Initialisierungsteil empfangsbereit eingerichtet

```
theReceiver.Create(PORT);
```

```
theReceiver.Listen();  .
```

Die PORT-Nummer muss beim Server-Socket fest vorgegeben werden (> 1024), da sich der Client-Socket genau mit dieser Port-Nummer verbinden muss.

In der CRcv-Klasse ist dann zu definieren, was der Server-Socket leisten soll. Dazu werden verschiedene in der Basisklasse schon vorhandene Methoden neu implementiert. Mit dem *Class Wizard* ('Add Function') werden die Funktionsgerüste für `OnAccept()`, `OnReceive()` und `OnClose()` erstellt. `OnAccept()` hat die Aufgabe, den Server-Socket zu duplizieren und die vom Clienten ankommende Verbindungsanfrage mit dem Duplikat anzunehmen:

```
void CRcv::OnAccept(int nErrorCode) {
  if (nErrorCode)
    CSocket::OnAccept (nErrorCode);
  CRcv * wsWork = new CRcv();
  if (!Accept(*wsWork))
    AfxMessageBox ("Accept: Fehler");
}  .
```

Durch die Duplizierung bleibt der Original-Socket empfangsbereit für weitere Anfragen.

Die zu verrichtende Aufgabe wird in `OnReceive()` festgelegt:

```
void CRcv::OnReceive(int nErrorCode) {
  if (nErrorCode)
    CSocket::OnReceive (nErrorCode);
  const int buflen = 2000;
  char Q[buflen];
  Q[Receive(Q, buflen)] = '\0';
  ...
  // action to be performed by the socket
}  .
```

Hier kann die mit der Funktion `Receive()` empfangene Anfrage `Q` analysiert und beantwortet werden oder mit

```
CString Peer;
UINT Port;
GetPeerName(Peer, Port);
```

festgestellt werden, woher die Anfrage kam, um nur bestimmte Clienten zuzulassen.

Das HTTP-konforme Versenden eine GIF-Datei an den Clienten (das könnte in diesem Fall ein Web-Browser sein) wäre beispielsweise mit

```
CFile F;
BYTE * fbuffer;
F.Open("Uniloggr.gif", CFile::modeRead);
UINT length = F.GetLength();
```

```
fbuffer = new BYTE[length];
F.Read(fbuffer, length);
CString S = "HTTP/1.0 200 OK\nContent-Type: image/gif\n\n";
Send (S, S.GetLength());
Send (fbuffer, length);
delete [] fbuffer;
```

zu bewerkstelligen.

Nach Verbindungsende sollte sich das Socket-Duplikat wieder verabschieden, z. B. mit

```
void CRcv::OnClose(int nErrorCode) {
  if (nErrorCode)
    CSocket::OnClose(nErrorCode);
  delete this;                    // suicide
}  .
```

Dabei wird davon ausgegangen, dass der Client-Socket die Verbindung aufgibt. Beendet der Server-Socket die Verbindung nach dem Versenden der Daten – das ist beispielsweise bei Web-Servern der Fall, wird `OnClose()` nicht implementiert; stattdessen wird das Socket-Duplikat am Ende der `OnReceive()`-Methode mit '`delete this`' entfernt.

Zumindest in der Testphase eines derartigen Programms ist eine ausgiebige Fehlerbehandlung zweckmäßig, das obige Fragment eines Web-Servers wäre entsprechend zu erweitern:

```
CFile F;
BYTE * fbuffer;
try {
  if (!F.Open("Uniloggr.gif", CFile::modeRead))
    throw ("CFile::Open: Fehler");
  UINT length = F.GetLength();
  fbuffer = new BYTE[length];
  if (length != F.Read(fbuffer, length))
    throw ("CFile::Read: Fehler");
  CString S = "HTTP/1.0 200 OK\nContent-Type: image/gif\n\n";
  if (Send (S, S.GetLength())==SOCKET_ERROR)
    throw ("Send: Fehler");
  if (Send (fbuffer, length)==SOCKET_ERROR)
    throw ("Send: Fehler");
  }
catch (char * err) { AfxMessageBox (err); }
delete [] fbuffer;  .
```

8.5.2 Client-Socket

Ein Client-Socket ist sehr viel einfacher zu implementieren, da er genau dort eingerichtet werden kann, wo er gebraucht wird und im allgemeinen mit den vorhandenen Methoden der Klasse auskommt. Typisch könnte etwa die Anweisungsfolge

```
CSocket Client;
char rbuffer[LENGTH];
Client.Create ();
Client.Connect (SERVER, PORT);
Client.Send (Q, Q.GetLength());
Client.Receive (rbuffer, LENGTH);
...
Client.Close();
```

sein, die eine Verbindung zu einem `SERVER` aufbaut und nach der Anfrage `Q` die Antwort in `rbuffer` erhält.

8.5.3 Mail-Client

Nur etwas komplizierter, in erster Linie umfangreicher, gestaltet sich die Erstellung eines Client-Programms, das einem feststehenden Protokoll folgen soll. Dies sei an einer einfachen Mailer-Klasse erläutert, die man beispielsweise dazu verwenden kann, die Nachricht vom erfolgreichen Ablauf eines Langzeitexperiments automatisch zu verschicken.

Zum Versenden von Mail im Internet steht auf den dafür eingerichteten Servern meist SMTP (*S*imple *M*ail *T*ransfer *P*rotocol), ein einfach zu bedienendes, weitgehend ungeschütztes Protokoll, zur Verfügung.

Die Protokollabwicklung wird in einer Klasse `CMailer` gekapselt, die `CSocket` erweitert:

```
class CMailer : public CSocket {
private:
  UINT SMTP;
  CString SENDER;
  CString SERVER;
public:
  CMailer() {
    SMTP   = 25;
    SENDER = "Your.Name";
    SERVER = "smtp.uni-osnabrueck.de";
  }
private:
  int Status();
  void Data(CString data, int result);
public:
```

```
  void Mail(CString to, CString subject, CString data);
};
```

Parameter, die in der Regel konstant bleiben (SMTP-Port, Absender, SMTP-Server), sind im Konstruktor fest eingestellt (über eine zusätzliche Konstruktorvariante lässt sich das auch flexibel halten). Das Versenden wird von der Funktion `Mail()` erledigt, die dazu die beiden privaten Hilfsfunktionen `Status()` und `Data()` benutzt.

`Status()` empfängt mit `Receive()` die Serverantworten und gibt der Statuscode des Servers zurück:

```
int CMailer::Status() {
  const int buflen = 500;
  char buf[buflen];
  buf[Receive(buf, buflen)] = '\0';
  return atoi(buf);
}  .
```

`Data()` verschickt Text mit Hilfe der `CSocket`-Methode `Send()` und überprüft den richtigen Antwortstatus:

```
void CMailer::Data(CString data, int result) {
  if (Send(data, data.GetLength())==SOCKET_ERROR)
    throw (CString("SEND DATA: Socket-Fehler"));
  if ( Status() != result ) {
    CString Error;
    Error.Format ("Status != %d: %s", result, data.Left(10));
    throw (Error);
}  .
```

`Mail` kreiert den Socket, verbindet mit dem Server, erledigt den gesamten formalisierten Eröffnungsdialog mit dem SMTP-Server, versendet die Daten (Datenende durch '.' am Zeilenanfang) und beendet danach die Verbindung:

```
void CMailer::Mail(CString to, CString subject) {
  CString S;
  if (!Create())
    throw (CString("Create: Fehler"));
  if (!Connect(SERVER, SMTP))
    throw (CString("Connect: Fehler"));
  if (Status() != 220)
    throw (CString("Connect: Status != 220"));
  Data("helo hi\n", 250);
  S.Format("mail from: %s\n", SENDER);
  Data(S, 250);
  S.Format("rcpt to: %s\n", to);
  Data(S, 250);
```

```
  Data("data\n", 354);
  S.Format("Subject: %s\n%s\n.\n", subject, data);
  Data(S, 250);
  Data("quit\n", 221);
  Close();
}  .
```

Fehlerhafte Statusrückmeldungen führen zum Abbruch, die jeweiligen Fehlermeldungen (`throw`) sollten - zumindest während der Testphase - im übergeordneten Programm mit `try` und `catch` abgearbeitet werden.

Die Verwendung der `CMailer`-Klasse wird durch ein Fragment aus dem übergeordneten Programm deutlich:

```
CMailer M;
try {
  M.Mail(Address1, Subject1, Data1);
  M.Mail(Address2, Subject2, Data2);
  }
catch (CString err) { AfxMessageBox(err); }
```

verschickt zwei Email-Nachrichten.

Literatur

Die Literaturhinweise sind – insbesondere die zitierten Lehrbücher betreffend – exemplarisch, willkürlich und zufällig, eine Vollständigkeit ist weder möglich noch angestrebt.

[1] H. Ibach, H. Lüth. *Festkörperphysik*. Springer, 1990.

[2] D. Geist. *Halbleiterphysik I: Eigenschaften homogener Halbleiter*. Vieweg, 1969.

[3] F. S. Goucher, G. L. Pearson, M. Spark, G. K. Teal, W. Shockley. *Theory and Experiment for a Germanium p-n Junction*. Phys. Rev. **81**, 637 (1951).

[4] W. Heywang. *Sensorik*. Springer, 1986.

[5] Analog Devices. *AD 592*. Datenblatt.

[6] G. Winstel, C.Weyrich. *Optoelektronik II – Photodioden, Phototransistoren, Photoleiter und Bildsensoren*. Springer, 1986.

[7] Burle Electronics. *Photomultiplier Handbook*. Burle, 1987.

[8] Burle Electronics. *Electro-Optics Handbook*. Burle, 1987.

[9] D. Geist. *Halbleiterphysik II: Sperrschichten und Randschichten – Bauelemente*. Vieweg, 1970.

[10] Hamamatsu Photonics. *Opto-Semiconductors, Condensed Catalog*. Datenblatt.

[11] Centronic Ltd. *High Performance Silicon Photodetectors*. Datenblatt.

[12] Ortel Vertriebs GmbH. *InGaAsP / InP-IR-LED's, InGaAs-Fotodioden (Telcom Devices Corp.)*. Datenblatt.

[13] M. J. E. Golay. *A pneumatic infrared detector*. Rev. Sci. Instr. **18**, 357 (1947).

[14] U. Tietze, C. Schenk. *Halbleiterschaltungstechnik*. Springer, 1991.

[15] Hermann Hinsch. *Elektronik: Ein Werkzeug für Naturwissenschaftler*. Springer, 1996.

[16] http://www.ni.com.

[17] http://www.agilent.com.

[18] http://www.mathworks.com.

[19] http://www.inria.fr.

[20] The MathWorks. *MATLAB Application Program Interface Guide*. Handbuch, 1998.

[21] http://sources.redhat.com/cygwin/.

[22] http://www.research.att.com/sw/tools/uwin/.

[23] http://www.mingw.org.

[24] http://www.bloodshed.net.

[25] http://www.mrc-cbu.cam.ac.uk/Imaging/gnumex20.html.

[26] Bei einer Standard-Installation:
<matlabroot>/help/pdf_doc/matlab/api/apiguide.pdf.

[27] Bei einer MATLAB-Standard-Installation findet sich das Inhaltsverzeichnis der API-Referenz in <matlabroot>/help/techdoc/apiref/apireftoc.html.

[28] Bei einer Standard-Installation:
<matlabroot>/help/pdf_doc/matlab/api/apiref.pdf.

[29] http://www.physik.uni-osnabrueck.de/kbetzler/gw/gw.html.

[30] Bässmann, Besslich. *Konturorientierte Verfahren in der digitalen Bildverarbeitung.* Springer Verlag, 1989.

[31] A. Savitzky, M. J. E. Golay. *Smoothing and Differentiation of Data by Simplified Least Squares Procedures.* Analytical Chemistry **36**, 1627–1639 (1964).

[32] W. Gander, J. Hřebíček. *Solving Problems in Scientific Computing using Maple and MATLAB.* Springer Verlag, 1995.

[33] Klaus Betzler. *Experimentsteuerung.* Vorlesungsskriptum, 1990.
http://www.physik.uni-osnabrueck.de/kbetzler/win32/es.pdf.

[34] Viktor Toth. *Visual C++ 5.* Markt & Technik, 1997.

[35] http://www.bbdsoft.com/.

[36] Brian W. Kernighan, Dennis M. Ritchie. *Programmieren in C.* Hanser, 1990.

[37] http://www.physik.uni-osnabrueck.de/kbetzler/win32/ansi/cpp.htm.

[38] http://www.ieee.org/.

[39] http://www.fapo.com/ieee1284.htm.

[40] http://www.physik.uni-osnabrueck.de/kbetzler/win32/ieee1284/1284int.htm.

[41] Georg Walz. *Grundlagen und Anwendung des IEC-Bus.* Markt & Technik, 1982.

[42] Lothar Preuss, Harald Musa. *Computerschnittstellen : Dokumentation der Hard- und Software mit Anwendungsbeispielen CENTRONICS, IEC-BUS, V.24.* Hanser, 1989.

[43] Hewlett Packard. *HP Standard Instrument Control Library: User's Guide for Windows.* Handbuch, 1996.

[44] Hewlett Packard. *HP Standard Instrument Control Library: Reference Manual*. Handbuch, 1996.

[45] http://www.usb.org.

[46] http://www.usbstuff.com.

[47] http://www.physik.uni-osnabrueck.de/kbetzler/Win32/usb/usbspec.pdf.

[48] http://www.ietf.org/rfc/rfc1700.txt.

[49] http://www.ietf.org/.

[50] David J. Kruglinsky. *Inside Visual C++ Version 5*. Microsoft Press, 1997.

[51] http://www.physik.uni-osnabrueck.de/kbetzler/TexCo/colormap.pdf.