



**Eine Einführung ins Programmieren von Anfang an**

**Teil 3: Programmieren unter Windows/Linux  
(Unvollständig, nur zwei Überblickskapitel)**

Christof Schatz

Email: [schatz@askos.de](mailto:schatz@askos.de)

[www.askos.de](http://www.askos.de)

pdf-Version der Website [www.askos.de/tutorial](http://www.askos.de/tutorial), Stand Okt. 2005

## **Teil 3: Einfaches Programmieren unter Windows/Linux**

[Ein Blick aus dem Fenster](#)

Seite 4

[Moderne Betriebssysteme](#)

Seite 7

---

# Ein Blick aus dem Fenster

---

Vielleicht hatten Sie anfangs, als hier von "DOS" und "BASIC" die Rede war, die Nase gerümpft. Und jetzt, nachdem Sie die letzten Kapitel in die Möglichkeiten eines BASIC-Entwicklungssystems der 80er-Jahre hineinbegleitet haben, wollen Sie vielleicht gar nicht mehr weg davon. Nun, Sie müssen auch nicht. Aber eines kann ich Ihnen versichern: In die riesige Welt der Informatik haben Sie gerade einmal einen zaghaften Blick geworfen. Da draussen warten noch Möglichkeiten, die Sie noch gar nicht erahnen!

Inzwischen wissen Sie sehr gut, was "Programmieren" heisst. Sie haben auch ein anderes Verständnis für den Computer entwickelt. Wahrscheinlich arbeiten Sie inzwischen auch mit Ihren Anwendungsprogrammen anders, da Sie nun ein bisschen nachvollziehen können, wie der Computer "denkt" und wie diese Programme von den Grundprinzipien her geschrieben sind, die Sie benutzen.

Wir haben die einfache Programmierung hinter uns gelassen und schon einige Teile der "Strukturierten Programmierung" gelernt. Allerdings fehlt hier noch viel. Desweitem wissen wir noch nichts von der s.g. "Objektorientierung" (OO), auf der heute alle modernen Entwicklungssysteme und -sprachen aufbauen. Ausserdem können wir bisher noch kein Programm schreiben, das "anständig" aussieht wie andere Programme auch: Mit Menüs, Knöpfen und Fenstern. Ja, und der eine oder andere würde sicher gerne mal ein tolles Spiel schreiben, so etwas wie Half Life, Quake oder Doom. Überhaupt: Wie erzeugt man einen anständigen Sound, wie spiele ich mal ein Video ab?

Wir müssen hier zwischen zwei weiterführenden Richtungen unterscheiden:

## A. Der Weg zum modernen PC

Der ist gar nicht so lang wie es im Moment scheinen mag. Es gibt heute eine Hülle und Fülle moderner BASIC-Implementierungen, alle mehr oder weniger ähnlich zu QBASIC, aber für die heute modernen Betriebssysteme geschrieben. Die Ansteuerung von Fenstern, schneller Grafik, Stereo-Sound usw. ist mit den meisten von ihnen kein Problem. Genauso wie in unserem 3D-Beispiel gibt es da einfach eine grosse Zahl neuer Funktionen. Nur, dass wir die nicht alle selbst programmieren müssen, sondern wir bekommen sie fertig serviert und müssen nur lernen, damit umzugehen. Das Einzige prinzipiell Neue daran können die oben schon erwähnten "Klassen" und "Objekte" sein. ("Klassen" sind die Typen von "Objekten"). Solange man aber keine neuen programmieren muss, sondern nur die bereitgestellten benutzen will, wird man sich auch ohne tieferes Verständnis damit arrangieren können.

Das wichtigste Hilfsmittel, sich in solche Funktionen- und Objektsammlungen einzuarbeiten, sind *Beispielprogramme*. Bei jeder Entwicklungsumgebung sind Beispiele dabei. Schauen Sie sich diese ausführlich und genau an; hier können Sie sehr viel lernen. Oft geht es sehr viel schneller, sich das Verständnis aus einem Beispielprogramm zu holen als eine komplizierte Dokumentation durchzukaufen - vorausgesetzt, man beherrscht die Grundzüge der Programmierung. Das tun wir schon.

## B. Der Weg zu modernen Programmiersprachen

Man kann also moderne Programme für einen modernen PC in BASIC schreiben - kein Problem. Dieser Teil wird sich dieser Aufgabe widmen. Aber die Grösse und Komplexität der Programme und Projekte, die Sie erstellen, wird durch die Möglichkeiten von BASIC begrenzt werden. Es wird einfach so sein, dass bei einer bestimmten kritischen Grösse des Programms trotz aller Strukturierung die nächsten 100 Zeilen Sie wochenlange Arbeit kosten werden, weil diese 100 Zeilen mit fast allen der restlichen 10000 Zeilen zusammentreffen können und dann soviel passieren kann, dass Sie aus dem Debuggen gar nicht mehr raus kommen. An diesem Punkt werden Sie entdecken, dass es vielleicht doch gut ist, nach anderen, moderneren Programmiersprachen Ausschau zu halten.

Ausserdem können Sie auch schon bei einigen Hundert Zeilen von einer modernen Programmiersprache sehr profitieren. Ich lerne aller vier bis fünf Jahre eine neue Programmiersprache, manchmal auch öfter, je nach Bedarf der Projekte, mit denen ich es zu tun habe. Die letzte habe ich vor zwei Jahren angefangen. Sie hat die Eigenschaft, dass man damit eigentlich keine Laufzeitfehler mehr macht. Man programmiert, sagen wir, 100 oder 200 Zeilen, macht dabei ein paar Syntaxfehler, bereinigt diese und das Programm läuft perfekt. Falsche Initialisierungen, Index-Überläufe bei Arrays, gar falsche Variablenmanipulationen? Alles Fehlanzeige. Der Preis ist ein gewisser Lernaufwand für neue Konzepte. Aber genau das ist es ja auch, was Spass macht.

Nachdem wir in diesem Teil uns mit der Programmierung auf einem (halbwegs) modernen Betriebssystem beschäftigt haben, nebenbei noch unsere Kenntnisse in strukturierter Programmierung erweitern, werden wir uns im darauffolgenden Teil anhand einer neuen Programmiersprache systematisch mit der objektorientierten Programmierung befassen.

## **Zu diesem Teil**

Wie schon oben geschrieben: Es gibt heute sehr viele BASIC-Entwicklungssysteme, angefangen bei Skriptsprachen für die Systemsteuerung über eigenständige Anwendungsentwicklung über s.g. Applikationsprogrammierung bis hin zu Spieleentwicklungssystemen. Angesichts der Tatsache, dass BASIC hier eigentlich eher als Startrampe für modernere Sprachen dienen soll, sollen in diesem Teil nur exemplarisch zwei Systeme gestreift werden, ein freies Tool zur Anwendungsprogrammierung und ein Tool zur s.g. Office-Programmierung.

## **...und wo lerne ich meine Shooter zu programmieren?**

Die Spieleprogrammierung ist an sich eines der anspruchsvollsten Programmiergebiete überhaupt. (Und eines der reizvollsten). Wer von der Pike weg professionelle Spiele entwickeln will, der hat an dieser Stelle erst ein Tausendstel des notwendigen Lernweges zurückgelegt. Es gehören Abertausende von Stunden dazu, hier etwas Anständiges zuwege zu bringen. Allein geht sowieso gar nichts. Das heisst, man muss auch lernen, im Team zu programmieren.

Aber so schlimm ist die Lage für Anfänger denn doch nicht. Wenn man die Sache andersherum aufzieht und sagt: "Na, mal sehen, was denn für einen BASIC-Freak zu machen ist", der wird eher angenehm überrascht sein. In den letzten Jahren sind eine ganze Reihe spezieller Spiele Entwicklungssysteme herausgekommen, die sich an Anfänger richten. Zwei davon, "Blitz 3D" und "Dark Basic", erfordern auf Programmiererseite nur die Beherrschung von BASIC - Ihre Kenntnisse dürften zu diesem Zeitpunkt einigermaßen ausreichen dafür. Dazu kommt natürlich die Einarbeitung in die ganzen Editoren, Grafikprogramme, Formate usw. Auch hier werden Sie einige Zeit benötigen, reinzukommen - aber eher ein paar Wochen, keinesfalls Jahre...

[Blitzbasic](#)

[Dark Basic](#)

---

# Moderne Betriebssysteme

---

Während es in den 80er-Jahren im Wesentlichen nur ein Betriebssystem für PC's gab, nämlich DOS, entwickelten sich in den 90er-Jahren zwei Hauptzweige, die von der Geschichte und den Rahmenbedingungen unterschiedlicher nicht sein könnten: MS Windows und das, was ich mal als "Free Unix" bezeichne, der Oberbegriff von Linux und FreeBSD Unix. Daneben gab es natürlich schon in den 80er-Jahren die "Apple-Welt", die ich allerdings überhaupt nicht kenne und zu der ich daher auch nichts sagen kann.

Bis jetzt konnten Sie die eingesetzten Entwicklungswerkzeuge relativ problemlos auf allen drei Plattformen nutzen, da DOS unter Windows, Linux/FreeBSD und Mac OS X gleichermassen emuliert wird. Doch jetzt müssen wir uns entscheiden, ob wir uns der Programmierung unter Windows oder der unter UNIX widmen wollen. Oder doch nicht? Die Lage ist ein bisschen verwirrend und es soll versucht werden, in diesem Abschnitt eine Skizze zu erstellen, wie man sich das Zusammenspiel zwischen Entwicklungssystem und Betriebssystem vorstellen kann.

Als anschauliches Beispiel wählen wir uns dazu das Grafiksystem. Wir können den gleichen Aufbau aber auch auf das Thema "Netzwerk", "Drucker" oder "Filesystem" übertragen.

## Die Hardware- und Treiber-Ebene

Ein ganz guter Ausgangspunkt ist wieder das Uralt-Minimalbetriebssystem DOS. Wir fragen uns, wie wir unter DOS ein Programm schreiben müssten, das das gleiche leistet wie ein modernes Fensterprogramm, sei es unter Windows, Linux oder BSD. Nun, das ginge bei der Ansteuerung der Grafikkarte los. QBASIC liefert nur Funktionen zur Ansteuerung der VGA-Modi. Die eigentliche Grafikkarten-Hardware mit ihren grossen Framebuffern, 3D-Funktionen usw. liegt unter QBASIC zunächst lahm. Eine Grafikkarte ist wie ein Stück eigener Computer mit Speichern, CPU usw.. Und natürlich könnte man QBASIC so erweitern, dass es diese ganze Hardware ansteuert. Dazu müsste man die Grafikkarten-eigene Maschinensprache in QBASIC-Befehle übersetzen. Aber was für ein Aufwand! Und das für eine einzige spezielle Grafikkarte! Denn jede Grafikkarte hat natürlich ihre eigene Maschinensprache und Steuerungslogik.

Hier setzt nun die unterste Stufe des Betriebssystems ein. Es stellt eine Routinensammlung für einfache Grafikoperationen wie Punkte- und Linienzeichnen bereit. Der Grafikkartenhersteller wiederum stellt ebenfalls eine Routinensammlung bereit. Was in den Routinen vor sich geht, weiss nur der Grafikkartenhersteller, aber wie die Funktionsköpfe aussehen müssen, das bestimmt das Betriebssystem, denn dieses ruft diese Routinen auf. Man nennt diese Hardware-Hersteller-Routinen "Treiber". Der Programmier hat mit ihnen nichts zu tun, er ruft einfach die Linienzeichnenfunktion des Betriebssystems auf und erst diese ruft wiederum die Treiberroutine auf. Was wir also lernen: Um die spezifische Hardware brauchen wir uns nicht zu kümmern, wir müssen wissen, wie man diese Grafikroutinen des Betriebssystems aufruft. Diese nennen wir mal "Basis-Grafik". Unter Linux/BSD heisst diese "Basis-Grafik" "X-Windows", unter Windows "GDI" (Graphics Device Interface)(langsame Grafik), bzw. "Direct X" (schnelle Grafik).

## Die API-Ebene

Sind wir auf dieser "Basis-Grafik"-Ebene, dann können wir also z.B. die Farbtiefe, die Auflösung usw. bestimmen und die entsprechenden Linien, Punkte usw. ausgeben. Aber unsere Fenster müssen wir dann selbst entwerfen. Ausserdem ist es enorm schwierig, diese Basis-Grafik als Programmierer anzusteuern, an sich ist sie nur zur Benutzung durch das Betriebssystem selbst gedacht.

Ein Programm, das mit Fenstern, Menüs, Knöpfen usw. daherkommt, nennt man ein "GUI-Programm", GUI=Graphical User Interface, zu deutsch: Grafische Benutzerschnittstelle. Ein GUI- Betriebssystem erwartet von jedem Programm, dass es ein GUI mitbringt. Und das macht die Schwierigkeit aus, auf einem GUI-Betriebssystem das Programmieren zu lernen - selbst das einfachste "Hello World" braucht ein GUI. Es sei denn, man wählt den Notausgang in die Konsole, wie wir es getan haben. Aber das ist natürlich nicht im Sinne des Erfinders.

Die Eigenheit von GUI's ist es, dass sie *ereignisorientiert* sind. Ein Programm ist nicht mehr sequentiell aufgebaut, sondern es besteht aus einer Sammlung von Routinen, von denen eine jede die Antwort auf ein mögliches Ereignis darstellt - ein Mausklick, ein Tastendruck, das Ankommen einer Nachricht usw... Es läuft natürlich schon ein sequentielles Hauptprogramm ab, aber das Hauptprogramm eines jeden Programms ist - das Betriebssystem. Das Betriebssystem ruft also beim Eintritt von Ereignissen bestimmte Routinen des Anwenderprogramms auf. Dementsprechend "verschweisst" sind Anwenderprogramm und Betriebssystem.

Die Schweissnaht, das ist die s.g. API. (Application Interface). Das ist eine Sammlung von Datentypen und Routinen, die das Betriebssystem zum Aufbau von Fenstern, Menüs usw., aber auch zum Umgang mit Ereignissen bereitstellt. Wir müssen also nicht via line()-Befehl unsere eigenen Fenster entwerfen. Wir dürfen das nicht einmal. Der einzige Weg, ein Fenster in einem

GUI-Betriebssystem zu verwenden, ist, eine Routine "newwindow()" oder so ähnlich aufzurufen, die Bestandteil der API ist. Die API wiederum zeichnet dann alles mit Hilfe der Basis-Grafik und anderer Basisroutinen hin, die wiederum alles an die Hardwaretreiber weiterleiten, die wiederum die Hardware steuern. Wir sehen das Zwiebelprinzip: Von innen nach aussen wird es hardwareunabhängiger und inhaltsabhängiger.

Aber auch API-Programmierung ist ein hartes Geschäft. So sieht ein BASIC-Programm aus, das über die Windows-API "Hello World!" ausgibt:

```

''
''
''
''
defint a-z
option explicit
option private

'$include once:'win\kernel32.bi'
'$include once:'win\user32.bi'
'$include once:'win\gdi32.bi'

declare function      WinMain      ( byval hInstance as long, _
                                     byval hPrevInstance as long, _
                                     szCmdLine as string, _
                                     byval iCmdShow as integer ) as integer

''
'' Entry point
''
    end WinMain( GetModuleHandle( null ), null, Command$, SW_NORMAL )

'' :::::::
'' name: WndProc
'' desc: Processes windows messages
''
'' :::::::
defint a-z
function WndProc ( byval hWnd as long, _
                  byval message as long, _
                  byval wParam as long, _
                  byval lParam as long ) as integer

    dim rct as RECT
    dim pnt as PAINTSTRUCT
    dim hDC as long

    WndProc = 0

    ''
    '' Process message
    ''
    select case ( message )

        ''
        ''
        ''
        case WM_CREATE
            exit function

        ''
        '' Windows is being repainted
        ''
        case WM_PAINT

            hDC = BeginPaint( hWnd, pnt )
            GetClientRect hWnd, rct

            DrawText hDC, "Hello World!", -1, _
                rct, DT_SINGLELINE or DT_CENTER or DT_VCENTER

            EndPaint hWnd, pnt

            exit function

        ''
        ''
        ''
        case WM_KEYDOWN
            if( lobyte( wParam ) = 27 ) then

```



```

        PostMessage hWnd, WM_CLOSE, 0, 0
    end if

    ''
    '' Window was closed
    ''
    case WM_DESTROY
        PostQuitMessage 0
        exit function
    end select

    ''
    '' Message doesn't concern us, send it to the default handler
    '' and get result
    ''
    WndProc = DefWindowProc( hWnd, message, wParam, lParam )
end function

'' ::::::::::
'' name: WinMain
'' desc: A win2 gui program entry point
''
'' ::::::::::
defint a-z
function WinMain ( byval hInstance as long, _
    byval hPrevInstance as long, _
    szCmdLine as string, _
    byval iCmdShow as integer ) as integer

    dim wMsg as MSG
    dim wcls as WNDCLASS
    dim szAppName as string
    dim hWnd as unsigned long

    WinMain = 0

    ''
    '' Setup window class
    ''
    szAppName = "HelloWin"

    with wcls
        .style = CS_HREDRAW or CS_VREDRAW
        .lpfnWndProc = @WndProc
        .cbClsExtra = 0
        .cbWndExtra = 0
        .hInstance = hInstance
        .hIcon = LoadIcon( null, IDI_APPLICATION )
        .hCursor = LoadCursor( null, IDC_ARROW )
        .hbrBackground = GetStockObject( WHITE_BRUSH )
        .lpszMenuName = null
        .lpszClassName = strptr( szAppName )
    end with

    ''
    '' Register the window class
    ''
    if ( RegisterClass( wcls ) = false ) then
        MessageBox null, "This program requires Windows NT!", szAppName, MB_ICONERROR
        exit function
    end if

    ''
    '' Create the window and show it
    ''
    hWnd = CreateWindowEx( 0, _
        szAppName, _
        "The Hello Program", _
        WS_OVERLAPPEDWINDOW, _
        CW_USEDEFAULT, _
        CW_USEDEFAULT, _
        CW_USEDEFAULT, _
        CW_USEDEFAULT, _
        null, _
        null, _
        hInstance, _
        null )

```

```

ShowWindow    hWnd, iCmdShow
UpdateWindow  hWnd

''
'' Process windows messages
''
while ( GetMessage( wParam, null, 0, 0 ) <> false )
    TranslateMessage wParam
    DispatchMessage wParam
wend

''
'' Program has ended
''
WinMain = wParam.wParam
end function

```

Das Beispiel wurde den Examples von FreeBasic 0.13 entnommen.

Haben Sie was verstanden? Nein? Ich auch nicht...Ausser ganz oben: Die ersten Zeilen, die mit '\$include...' beginnen, adressieren die Routinen der API, die wiederum die drei wichtigen Kernteile von Windows ansprechen, namens "User", "Kernel" und "GDI". Von GDI haben wir ja hier schon einmal etwas gehört.

## Die Ebene der Runlib's und Komponenten

Eine Sammlung mit Routinen zum Allgemeingebrauch heisst "Bibliothek", englisch "Library" oder abgekürzt "Lib". Man muss zwei Formen von Bibliotheken unterscheiden: Bibliotheken, die Routinen in Maschinsprache, also in kompilierter Form enthalten. Und Bibliotheken, die Routinen vor der Kompilierung, also im Quelltext enthalten. Erstere nenne ich hier mal "Runlib's", einen offiziellen, betriebssystemübergreifenden Namen kenne ich gar nicht dafür. "Runlib" deshalb, weil sie fertig für den "Run" sind, nicht mehr kompiliert werden müssen. Unter Windows sind das die berühmt berüchtigten "DLL's", weil die entsprechenden Kompilate mit dem Suffix ".dll" versehen werden. Unter Linux haben sie die Endung ".so".

Runlib's sind an sich die schickere Variante. Sie haben zwei Vorteile: Man kann sie aus (fast) beliebigen Programmiersprachen heraus benutzen. Und sie kosten keine Compilerzeit. Allerdings sind sie schwieriger zu erstellen, zu benutzen und zu organisieren. Wenn z.B. ein Fehler innerhalb einer Runlib auftritt, ist der Debugger machtlos. Er kann nicht einfach auf der Quelltextzeile stehen bleiben, wo der Fehler aufgetreten ist, weil es ja gar keine Quelltextzeile gibt. Ein weiterer Nachteil ist, dass man Runlibs nicht modifizieren kann und dass man aus ihnen nichts lernen kann. Aus Quelltextbibliotheken hingegen kann man oft eine Menge lernen.

Runlib's sind die Verbindungstür für den Programmierer zwischen seinem Entwicklungssystem und dem Betriebssystem. Die Zahl der Runlib's ist sowohl unter Windows wie auch unter Linux schier unübersehbar geworden. Es gibt auch eine grosse Anzahl solcher Runlib's, die für beide Systeme verfügbar sind und zur Ansteuerung identische Funktionsköpfe haben. Dadurch kann man Programme entwickeln, die auf beiden Plattformen (oder sogar auch auf dem Mac) laufen.

Als Beispiel dazu sei hier die Bibliothek "Gtk" genannt. "Gtk" steht für "Gimp Toolkit", [siehe hier](#). Es umfasst nicht nur die Basisfunktionen zur Arbeit mit Fenstern (Fenster, Menüs, einfache Knöpfe und Textfelder) wie die Windows-API, sondern zahlreiche weiterführende Elemente wie Register, Lineale, Fortschrittsbalken, Timer, Zähler, Dialogfenster, komplette Datei-Öffnen-Dialoge, Druck-Dialoge usw. Man nennt solche Elemente oft "Widgets". Gtk gibt es in einer Version für Linux, eine für Windows und wahrscheinlich auch eine für Free BSD und für Apple-Mac. Programme, die also ausschliesslich Gtk benutzen, sind in der Regel auf allen diesen Plattformen kompilierbar. Ausser dass sie meist noch weitere und andere Dinge beherrschen als die elementaren Betriebssystem-API's, sind sie meist auch effizienter und einfacher anzusteuern als diese:

```

option explicit
#include once "gtk/gtk.bi"

#define NULL 0

dim as GtkWidget ptr win
dim as GtkWidget ptr message_dialog
dim as integer i
dim as zstring * 128 buf

gtk_init( NULL, NULL )

'' create a new window
win = gtk_window_new( GTK_WINDOW_TOPLEVEL )
gtk_widget_set_usize( win, 400, 200 )
gtk_window_set_title( win, "Hello World" )

buf="Hello World!"
message_dialog=gtk_message_dialog_new(win,0,GTK_MESSAGE_INFO,GTK_BUTTONS_OK,buf)
gtk_widget_show( win )

```

```
gtk_widget_show(message_dialog)

gtk_main( )

end 0
```

erzeugt folgenden netten Output:



(Allerdings ist im obigen Quelltext trotzdem noch nicht alles richtig - das Programm gibt seine Ressourcen nicht mehr zurück...)

## Die Ebene der Bindungen

Und wie nutze ich nun diese schicke Bibliotheken aus QBASIC heraus? Gar nicht. Denn QBASIC ist kein Windows- oder Linux-Programm, aber QBASIC stellt auch keine *Deklarationen* der ganzen Funktionsköpfe, Typen und Konstanten zur Verfügung, die man benötigt, um die Runlib's ansprechen zu können. Die Sammlung der Deklarationen nennt man die "Bindungen" der Programmiersprache an die Bibliotheken. Man kann also mit einem Entwicklungssystem all die Runlibs benutzen, die

- A. auf dem Rechner installiert sind und
- B. für die im Entwicklungssystem Bindungen existieren.

Im oberen Beispiel sieht man, welche Datei für die Bindung benötigt wird: `#include once "gtk/gtk.bi"`, also die Datei `"gtk.bi"` im Verzeichnis `"gtk"`. `"bi"` steht dabei schon für `"Binding"`.

## Runlibs, Komponenten und Bindungen

Bei einer grossen Zahl von Runlib's spricht man auch von "Komponenten" oder "Widgets" (siehe oben) oder "Controls". "Controls" und "Komponenten" hat sich dabei eher in der Windows-Welt etabliert, "Widgets" sagt man meistens in der UNIX-Welt. In der Linux-Welt z.B. gibt es zwei grosse Komponentensammlungen, Gtk und Qt, die mit praktisch allen Linux-Distributionen ausgeliefert und installiert werden. Sie gehören quasi fest zum Betriebssystem. Schon deshalb, weil die Benutzeroberfläche unter Linux gewählt werden kann: Man entscheidet sich entweder für KDE, dessen Programme auf Qt aufbauen, oder für Gnome, dessen Programme auf Gtk aufbauen. Wir sehen: Die "Hülle" des Betriebssystems besteht also aus Programmen, die Komponenten-Bibliotheken benutzen.

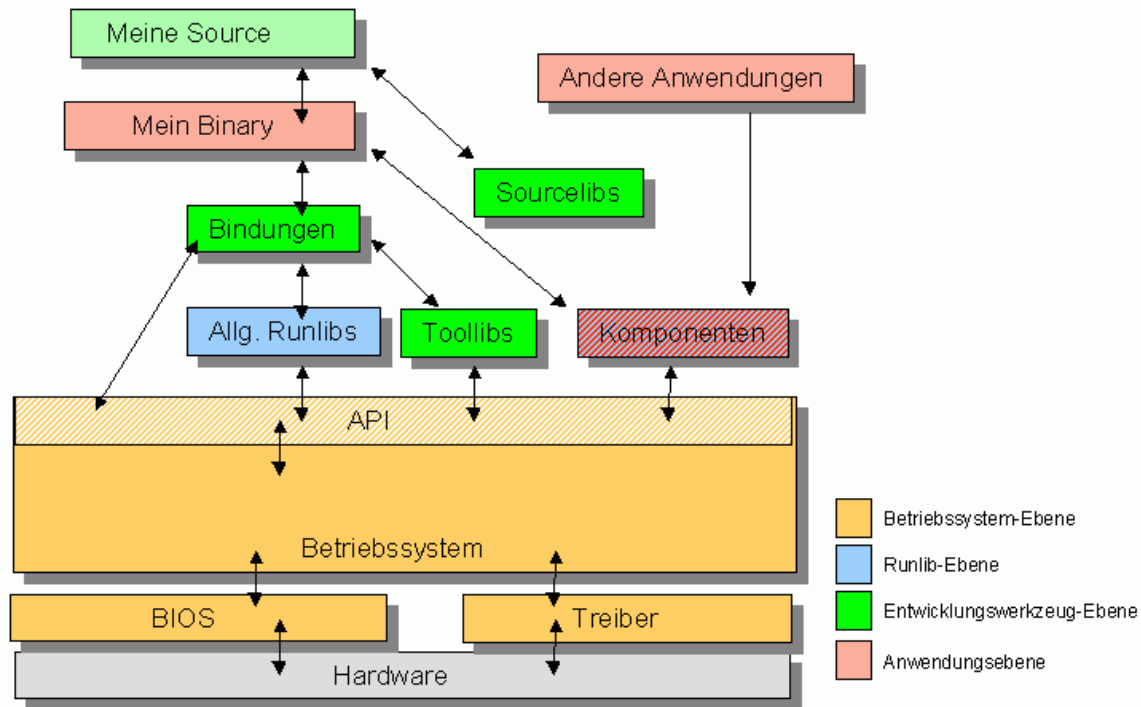
Auch bei Windows gibt es das. Unter Windows gibt es sogar ein System, mit dessen Hilfe Komponenten ohne spezifische Bindings benutzt werden können. Man nennt das "Active X". Jedes Programm kann solche "Active X"-Komponenten beisteuern und jedes Programm kann alle auf dem Rechner verfügbare Komponenten benutzen. Die Bündelung in "Bibliotheken" entfällt also hier. Das hat nicht nur Vorteile. Es hat eigentlich mehr Nachteile. Z.B. sind die ActiveX-Komponenten, die der Internet Explorer - der WWW-Browser der Firma Microsoft - mitbringt, nicht separat als Bibliothek erhältlich. Folglich müssen Programme, die diese Komponenten benutzen, zwingend voraussetzen, dass dieses spezielle Programm installiert ist - nur, damit eben auch dessen Komponenten installiert sind. Ähnlich ist das mit dem Media Player der gleichen Firma.

Aber natürlich gibt es unter Windows auch jede Menge "klassische" Bibliotheken. Die allerersten Bibliotheken unter Windows kamen von Microsoft selbst in Form der s.g. Windows SDK's (Source Development Kits). Dann wurden die MFC, "Microsoft Foundation Classes" für lange Zeit zum Standard. Die Konkurrenzfirma Borland etablierte die OWL-Bibliothek (Open Windows Library) und später die VCL, die Visual Component Library. Usw. Allerdings gab es für die meisten Entwicklungswerkzeuge der 90er-Jahre höchstens zwei Bindings: Eine an die werkzeugeigene Lib und - wenn das Werkzeug NICHT von Microsoft war - noch ein Binding an die MFC - und das meist nur in der teuren Professional-Variante.

Die Werkzeuge der 2000er-Jahre zeichnen sich - auch unter Windows - durch grössere Universalität aus. Zwar liefert der Hersteller oft nur die Bindings für ein oder zwei Bibliotheken mit, aber Drittanbieter und freie Programmierer stellen oft schnell Bindings zu allen gängigen Bibliotheken her. Unter Linux ist das universale Entwicklungswerkzeug der gcc und es gibt wohl fast keine Bibliothek unter Linux, die nicht ein Binding zum gcc hätte (mit Ausnahme der Java- und

.NET-Plattformen). Aber auch zu den anderen Werkzeugen, wie fpc oder den Skriptsprachen gibt es zahllose Bindings an alle gängigen Bibliotheken. Unter Windows sind Visual Studio und Delphi (neben Java, aber das ist eine eigene Welt) die grossen Entwicklungswerkzeuge und auch hier ist so gut wie alles benutzbar (sofern man es ggf. gekauft hat...) Daneben gibt es auch hier zahlreiche freie Entwicklungswerkzeuge mit jeweils einer Menge Bindings, z.B. DJGCC, Perl, Python, Ruby, Free Pascal und eben FreeBasic, das wir gleich noch etwas näher kennenlernen werden.

### Zusammenfassung: Ein moderner PC ist ein Hamburger



Wenn man als Programmierer also einen modernen PC vor sich hat, so ist das ein Gebilde aus vielen verschiedenen Lagen von Software und erst ganz unten kommt die Hardware. Dabei sind die meisten Lagen nichts anderes als Bibliotheken, die der Programmierer ggf. auch ansteuern kann. Wenn ich anfangen, zu programmieren, muss ich mich erstmal für ein Entwicklungswerkzeug entscheiden. Wenn ich das getan und mir dieses installiert habe, dann kann ich loslegen und den Source-Code schreiben (hellgrün). Das Entwicklungstool (grün) gibt mir schon eine Reihe von Sourcecode an die Hand den ich benutzen kann, teils gleich mitinstalliert, teils im Internet zu finden. Dies können Bibliotheken im Sourcecode sein, so wie Sie viele auf [www.qbasic.de](http://www.qbasic.de) finden. Oder Beispielprogramme. Oft installiert das Entwicklungswerkzeug auch eine oder mehrere eigene Runlibs, die ich hier mal Toollibs genannt habe. Und es installiert Bindungen, mit deren Hilfe mein Programm diese Libs benutzen kann. Und andere Runlibs (blau), die zur allgemeinen Verfügung stehen. Diese kann ich entweder separat installieren oder sie sind ohnehin schon im Betriebssystem vorhanden. Weiterhin finde ich auf meinem PC eine ganze Reihe von Komponenten vor, die ich benutzen kann, einfach so, ohne, dass dies das Entwicklungswerkzeug speziell vorhersehen muss. Als vierte Möglichkeit habe ich, falls das Entwicklungswerkzeug entsprechende Bindungen mitliefert, die API meines Betriebssystems direkt anzusteuern.

Was auch immer ich ansteuere, die Komponenten, Runlibs, Toollibs und Sourcelibs werden letztendlich die API benutzen, welche den Betriebssystemkern benutzt, welcher BIOS und Treiber benutzt, um schliesslich auf der Hardware das zu realisieren, was realisiert werden soll.

Man sieht also:

- Bei der modernen Programmierung hat man keine geschlossene überschaubare Anzahl von Befehlen mehr vor sich wie bei QBASIC. Sondern man ist mit unüberschaubar vielen Sammlungen an Routinen und Systemen konfrontiert.
- Das Entwicklungswerkzeug ist nicht alles. Hat man sich für eine Sprache und eine IDE entschieden und dort eingearbeitet, dann ist das erst ein kleines Stück des Wegs zum souveränen Programmierer. Vielmehr muss man die Bibliotheken "lernen". Und das ist viel mehr Arbeit!

### Die Versuchung der Libs

Es ist vor allem oft gar nicht so einfach, zu entscheiden, welche Bibliothek man nutzen soll. Hier gibt es allerdings Prioritäten. Liefert die Sprache Standardlibs mit, so sind diese als erstes zu benutzen. Ist dort nicht das dabei, was man sucht, so ist es für

einen Anfänger ganz gut, erstmal die Libs zu durchsuchen, die das Werkzeug mitbringt (Toollibs). Der Profi hingegen sollte eher dazu tendieren, Runlibs zu benutzen, die möglichst weit verbreitet sind. Erst, wenn man dann die Funktionalität noch immer nicht gefunden hat und es unverhältnismässig grosser Aufwand wäre, die Funktionen selbst zu programmieren, sollte man "exotischere" Bibliotheken heranziehen. Warum? Nun, erstens sollte man vermeiden, seinen und vor allem den PC derjenigen zu "vermüllen", denen man sein Programm weitergibt. Das abschreckende Beispiel ist ein Programm, das man auch hätte mit QBASIC programmieren können, das aber die Installation von vier oder fünf Bibliotheken erfordert, bevor man es starten kann. Das ist kein Programm, das ist ein Ärgernis.

Zum anderen wächst mit der Anzahl der Abhängigkeiten des eigenen Programms auch seine Gefahr, inkompatibel zu werden. Bei jedem Update einer Bibliothek, die das eigene Programm benutzt, besteht die Gefahr, dass es nicht mehr laufen wird. Ob so ein Update durchgeführt wird, bekommt man oft gar nicht mit, da dies vielleicht mit der Installation eines anderen Programms geschieht.

Schliesslich hat die Sache auch noch einen Hardware-Aspekt. Bindet man zuviele Bibliotheken ein, dann steigt der Speicherverbrauch des Programms unnötig an. Haben Sie sich mal gefragt, warum ein Programm, das nichts anderes tut als ein paar Icons anzuzeigen, auf die man klicken kann, 4 Megabyte RAM benötigt? Weil es eine Unzahl unnötiger Libs eingebunden hat.

Eine grosse Gefahr besteht oft darin, dass man sich durch die Entdeckungsreisen durch die Libs von der eigentlichen Programmieraufgabe ablenken lässt. Das gilt besonders für Grafik/Sound/Multimedia-Libs. Wenn man gerade rumspielt, ist das das eine, aber wenn man ein Anwendungsprogramm erstellt, sollte man sich immer fragen, ob für dieses Programm überhaupt eine Lib notwendig ist oder ob die Konsole als Schnittstelle nicht auch reicht. Und wenn, welche Bedienelemente wirklich förderlich sind und welche nur Spielerei sind.

Die Frage, wieviel Libs man benutzt, welche Aufgaben man mit eigenem Code erledigt, (was keineswegs immer optimal ist, da eigener Code immer fehleranfälliger und schwerer zu verstehen ist als öffentlicher Code), auf welche Funktionen man vielleicht auch ganz verzichten kann, das ist einer der Felder, die die Kunst und den persönlichen Stil des Programmierens ausmachen. Hier sei dazu erstmal nichts mehr gesagt, bevor wir nicht im nächsten Teil aktiv lernen, mit grösseren Projekten umzugehen.

## **Plattformen: Java und .NET**

Obwohl wir nun doch in diesem Kapitel sehr viel neue "Architektur", ganze Kathedralen voller Routinen und Ressourcen am Horizont auftauchen gesehen haben, ist das im Jahr 2005 und danach ein veraltetes Bild. Unter "Linux" oder "Windows" zu programmieren, ist zwar moderner als unter "DOS" zu programmieren, aber "State of the Art" ist das schon seit geraumer Zeit nicht mehr.

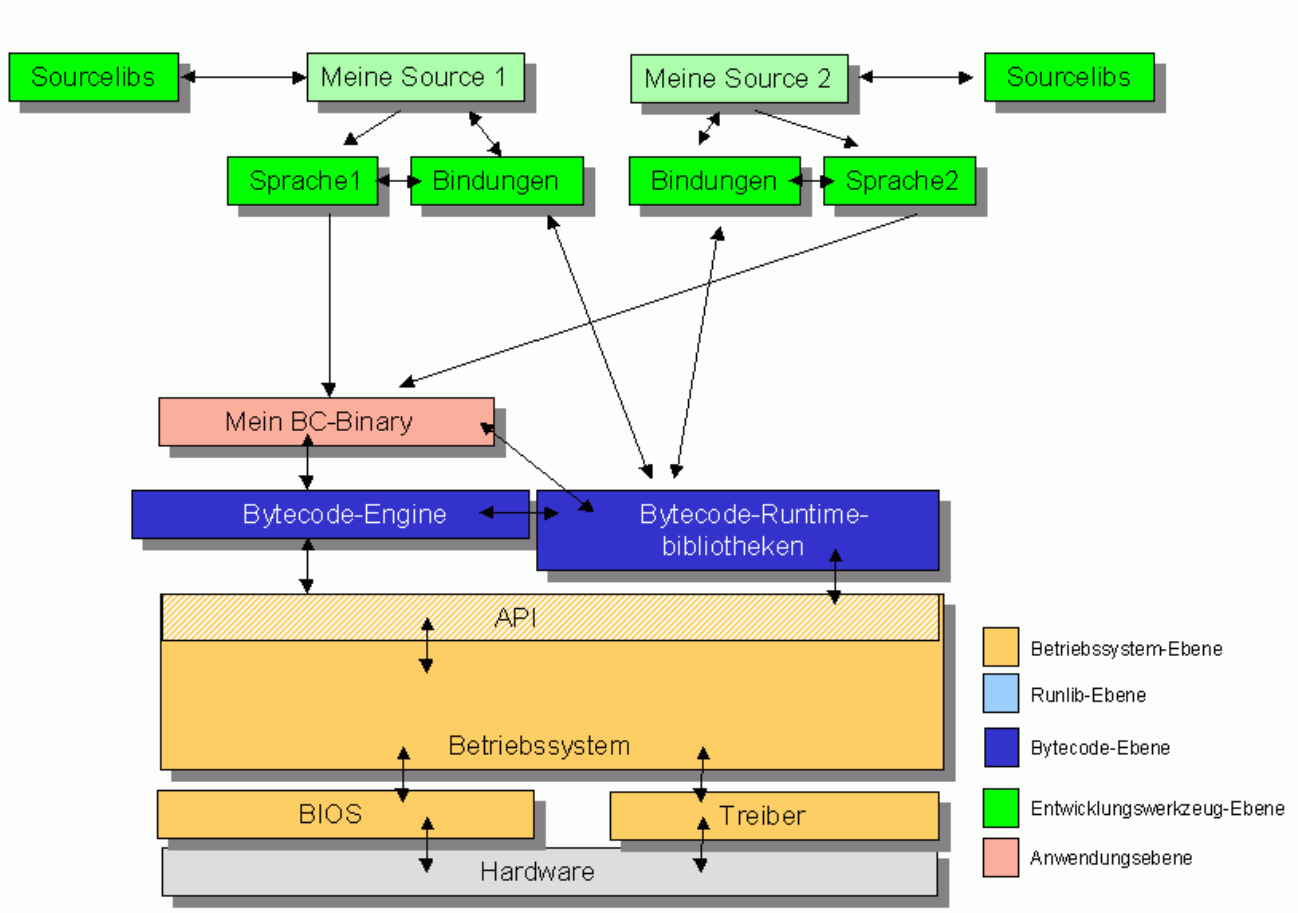
Wir nennen die Basis-API, auf die alle Runlibs aufsetzen, die "Plattform", unter der wir programmieren. Unter Linux ist das die UNIX-API, (grosse Teile davon nennt man auch "POSIX"), unter Windows die Windows-API (genauer: Win32-API). Es gibt allerdings noch zwei weitere Plattformen. Eine davon entstand 1995 in Verbindung mit dem Web und dürfte heute neben der Windows-API die am weitesten verbreitete Plattform sein: Java. Die andere wurde und wird von der Firma Microsoft entwickelt und soll für Windows die Plattform der Zukunft werden. Sie heisst ".NET". (Ausgesprochen: "Dotnet").

Java und .NET fügen in den PC-Hamburger von oben noch eine Schicht Käse ein. Der Sinn des Ganzen: Die Programmumgebung soll erstens standardisiert werden und zweitens soll das Betriebssystem und der PC vom Programm abgeschirmt werden - aus Sicherheitsgründen. Die zusätzliche Schicht dient also als strenger Filter und Konverter. Sie liegt gleich oberhalb der Betriebssystem-API und man könnte sie im allgemeinen BCI, ein "Byte Code Interpreter", nennen. Es ist tatsächlich ein Interpreter, der einen Quelltext in einer plattformspezifischen Sprache interpretiert, den "Bytecode". Die eigentlichen Sprachwerkzeuge kompilieren oder interpretieren nicht mit Maschinencode als Output, sondern sie erzeugen Bytecode. Mithin kann also der BCI in letzter Instanz noch kontrollieren, was da eigentlich passiert und ggf. Zugriffe sperren.

Neben dem BCI ist der zweite, fast noch wichtigere Teil die Plattform-Bibliothek. Richtig, es gibt nur eine Plattform-Bibliothek, keine werkzeug- oder sprachspezifischen Bibliotheken mehr. Und diese Bibliothek ist standardisiert. Das heisst, kommt ein Programm auf einen neuen PC, dann findet es nicht eine zufällig zusammengewürfelte Bibliothekslandschaft vor, sondern es liest nur die Versionsnummer der Plattform und weiss dann bis auf's Byte genau, was an Routinen, Komponenten usw. zur Verfügung steht.

Als kleinen Nebeneffekt bringt diese Herangehensweise den sofortigen Zugriff auf sehr reichhaltige Bibliotheken mit sich. Alles, was man sich sonst oft mehr oder weniger mühsam zusammenladen muss, ist auf einen Schlag da: Grafik, Sound, Datenbanken, mächtige String- und Listenverarbeitung, usw.. Allerdings ist dadurch auch der Einarbeitungsaufwand gewaltig. Nicht nur die Einarbeitung in die jeweils zentralen Sprachen fällt an, sondern vor allem die Einarbeitung in die Plattformenbibliotheken. Java oder .NET lernt man nicht mal so nebenbei in den Sommerferien. Anders gesprochen: Man hat von diesem Plattformen nur dann etwas, wenn man sie gründlich erlernt und die Mächtigkeit dieser Tools auch auszunützen weiss. Das setzt dann auch ein gutes Wissen über Objektorientierung voraus. Sonst kommt man über den herkömmlichen Weg a la Runlibs, Komponenten und Betriebssystem schneller ans Ziel.

Es gibt noch einen dritten Haken bei Plattformen: Nicht jeder hat jede Plattform installiert. Die meisten werden die Java Runtime Engine auf ihrem PC haben, da man sie für viele Webseiten benötigt und sie dann in der Regel automatisch lädt. Aber noch relativ wenige werden die .NET-Runtime geladen haben. Mit der nächsten Windows-Version 2007 ("Longhorn") wird zwar die .NET-Runtime automatisch installiert und geladen sein, aber dann entsteht wieder eine hässliche Inkompatibilität zwischen "altem" und "neuem" Windows. Ganz abgesehen davon, dass das Programm unter Linux wiederum gar nicht oder nur mit Mühe (Installation von "Mono") zum Laufen gebracht werden kann, während heute schon viele herkömmliche Entwicklungswerkzeuge auf Quelltextebene ziemlich plattformunabhängig und die entsprechenden Interpreter/Compiler leicht und kostenlos verfügbar sind.



Zusammenfassend: Plattformen werden sich wahrscheinlich langfristig durchsetzen, aber um kurz einmal ein paar hundert Zeilen Skript oder Simulationsprogramm oder Matheprogramm zu schreiben, sind sie derzeit eher "Overkill".