Entwickelt von Mark Sibly unter Verwendung von HiSoft's DevPac2..
Druck: UniPrint, Auckland, New Zealand.
Diese Handbuch wurde mit Soft Logik Page Stream2 & Page Liner erstellt.

Zusätzliche Übersetzungen, Vervollständigung und Bearbeitung für das PDF Format mit verlinktem Inhaltsverzeichnis von Jens Henschel

Inhalt

Vorwort

Herzlich willkommen im Kreise der Blitz2-Benutzer. Wir von Acid Software hoffen, Ihnen mit diesem System eine Umgebung zur Verfügung zu stellen, in der sie alle Ihre Ideen auf dem Amiga realisieren können.

Die größte Errungenschaft von Blitz2 sind die NewTypes. Das sind Strukturen, die wir aus C übernommen und so in BASIC installiert haben, als gehörten Sie immer hierher. Für Anhänger der strukturierten Programmierung gibt es Prozeduren und Funktionen. Der erweiterte Satz von Kontrollstrukturen, der in den vergangenen Jahren aus Pascal in BASIC eingebracht wurde, wird ebenso unterstützt.

Verkettete Listen erfreuen sich immer größerer Beliebtheit, deshalb haben wir sie in Blitz2 aufgenommen. Wir haben es sogar geschafft, sie schneller als gewöhnliche Felder arbeiten zu lassen.

Bei der Erstellung von Blitz2 haben wir sehr viel Mühe auf die volle Unterstützung des Betriebssystems verwandt. Menüs, Windows, Gadgets, und Screens sind alle als 'Objekte' in Blitz2 verfügbar. Natürlich unterstützt Blitz2 auch die gesamte Amiga Library, sodaß Betriebssystem-Aufrufe einfach nur wie Blitz2-Befehle parametriert werden.

Für Geschwindigkeits-Fanatiker gibt es neue Verfahren des 'Blitting'. Hintergrundgraphiken können durch spezielle Pufferung sauber restauriert werden und Schablonen können in mehreren Ebenen übereinandergelegt werden.

Wenn Sie das Blitz2 User Magazin abonnieren, erhalten Sie eingehendere Dokumentation zu diesen Themen sowie ständig erweiterte Bibliotheken. Der Befehlssatz von Blitz2 nimmt weiter zu, während wir unserem Ruf für 'Blitz'-BASIC gerecht werden.

Das erste aber, was Sie machen sollten, ist uns Ihre ausgefüllte Registrierungs-Karte zurückzusenden. Nur als registrierter Benutzer erhalten Sie die Updates von Blitz2 und können sicher sein, immer auf dem neusten Stand der Entwicklung zu sein.

Simon Armstrong

Der Gebrauch dieses Handbuchs

Diese Handbuch stellt alle notwendigen Informationen für die Benutzung von Blitz2 zur Verfügung. Es ist als Ergänzung zum Blitz2 Referenz-Handbuch gedacht, in dem sämtliche Befehle ausführlich beschrieben werden.

Das Benutzer-Handbuch enthält wichtige Informationen über den Gebrauch des Editor/Compilers, Erläuterungen der BASIC-Programmierung und weitgehende Erklärungen vieler neuer Konzepte, die zur Verbesserung von Blitz2 hinzugefügt worden sind.

Kapitel 1. Installation: Diese Kapitel liefert eine kurzen Überblick über Blitz2, Installations-Anweisungen und Hinweise zur Inbetriebnahme.

Kapitel 2. Grundlegendes über Blitz2: Dieses Kapitel wendet sich an Benutzer ohne Programmiererfahrung und liefert eine schrittweise Einführung in die Programmiersprache BASIC.

Kapitel 3. Typen, Felder und Listen: Dieses Kapitel beschreibt im Detail, wie Blitz2 Variablentypen, Strukturen, Zeiger, Felder und Listen verwaltet.

Kapitel 4. Prozeduren: Dieses Kapitel beschreibt, wie Prozeduren in Blitz2 implementiert werden.

Kapitel 5. Beispiele: Dieses Kapitel enthält eine Vielzahl von Programmbeispielen mit detaillierter Beschreibung ihrer Funktionsweise.

Kapitel 6. Fehlermeldungen & der Debugger: Dieses Kapitel enthält Informationen über Übersetzungs- und Laufzeitfehler und eine vollständige Beschreibung des Blitz2-Debuggers.

Kapitel 7. Objekte: Dieses Kapitel behandelt den Gebrauch von Objekten und liefert eine kurze Beschreibung der Haupt-Objekte, die von Blitz2 verwendet werden.

Kapitel 8. Blitzmodus: Dieses Kapitel erläutert den Blitzmodus, was er ist, was er tut und wie man ihn benutzt.

Kapitel 9. Beispiele für den Blitzmodus: Dieses Kapitel enthält einige Beispiele zur Erläuterung des Blitzmodus.

Kapitel 10. Weiterführende Themen: Enthält eine Vielzahl von Themen, die Blitz2 zu dem mächtigsten BASIC des Amiga Computers machen.

Anhang 1. Der Blitz2-Editor: Eine vollständige Beschreibung von Ted, dem Blitz2-Editor.

Anhang 2. Tastatur/-kürzel: Zum Gebrauch mit Ted.

Anhang 3. Programmier-Techniken: Eine eingehende Diskussion wertvoller Methoden, um bei großen Projekten die Übersicht zu behalten.

Anhang 4. ??????????????????

Anhang 5. Blitz2-Operatoren: Eine Liste der Blitz2-Operatoren.

Danksagungen an:
Mark Sibly, den Verfasser des Blitz2-Compilers, der Bibliotheken und der Dokumentation.
Rod Smith für die Gestaltung, den Blitz-Mann und die Graphiken auf der DemoDisk2.
Simon Armstrong für die Dokumentation und verschiedene Routinen.
Paul Andrews für die Spiele auf DemoDisk2.
Roger Lockerbie für Schreibarbeiten
Rich Parrill für die Repräsentation der Benutzer in den USA.
Aaron Koolen für verschiedene Beiträge.
Blitz One-Benutzer für Rückmeldungen und Unterstützung.

Kapitel 1

Installation

- Was Sie alles erhalten haben
- Blitz-Start
- Alles klar ?
- Die Beispiele
- Das aktuelle Verzeichnis
- Zusammenfassung

Was Sie alles erhalten haben

Programmdiskette
Diese Diskette enthält die drei Hauptdateien, die für Blitz2 benötigt werden:

Blitz2 - der Compiler
Ted - der Editor
DefLibs - Die Befehls-Bibliothek von Blitz2

Für Benutzer von Floppy-Systemen ist das Programm mit einer eingeschränkten Workbench1.3-Umgebung ausgerüstet worden, sodaß das Betriebssystem von der Diskette gestartet (gebootet) werden kann.

Beispieldiskette
Diese Diskette enthält eine Menge Demo-Programme und Beispiele, die mit Blitz2 geschrieben wurden. Diese Beispiele bieten eine Fülle an Informationen für Blitz2-Programmierer, sobald diese mit den wesentlichen Konzepten vertraut sind.

Benutzerhandbuch
Das Benutzerhandbuch bietet eine allgemeine Einführung in die Programmierung mit Blitz2. Außerdem werden die vielfältigen neuen Programmierkonzepte, die wir in diese Sprache eingeführt haben, erläutert.

Referenzhandbuch
Das Referenz-Handbuch enthält eine detaillierte Beschreibung sämtlicher Blitz2-Befehle mit kurzen Beispielen. Im Anhang werden die Fehlermeldungen des Compilers erläutert, und andere nützliche Informationen gegeben.

Registrierungskarte
Bitte füllen Sie diese Karte aus und senden sie Sie zurück an Acid Software. Sie erhalten nur dann Support, Fehlerkorrekturen und Aktualisierungen, wenn Sie als Blitz2 -Benutzer registriert sind. Wir würden gern von Ihnen wissen, zu welchem Zweck Sie Blitz2 verwenden und freuen uns über Verbesserungsvorschläge.

Verschiedenes
Darüberhinaus sollten Sie eine Ausgabe des Blitz2 User Magazines (sofern verfügbar) und ein oder zwei PD Spiele, die in Blitz2 geschrieben sind, erhalten haben. Im Blitz2 User Magazine befinden sich noch mehr Bespiele und weitere Dokumentation, nebst einer Diskette mit zusätzlichen Kommandos und Anwenderprogrammen.

Abonnieren lohnt sich!!

Blitz-Start

Zu allererst sollten Sie eine Sicherheitskopie der Disketten anlegen und die Originale sicher aufbewahren.

Benutzer von Diskettensysteme
Der einfachste Weg, die Disketten zu kopieren ist, von der Blitz2 Programmdiskette zu booten, das Symbol für die Blitz2-Diskette anzuklicken und dann *DUPLICATE* im Workbench-Menü auszuwählen. Detaillierte Angaben hierzu befinden sich im AMIGA-Handbuch.

Nach dem duplizieren der Blitz2 Programmdiskette wird die Kopie wahrscheinlich den Namen „Copy of Blitz2“ besitzen. Wenn das der Fall ist wählen Sie das Symbol für die Diskette an (indem Sie es anklicken), wählen *RENAME* im Workbench-Menü aus und benennen die Diskette in „Blitz2“ um.

Den selben Vorgang wiederholen Sie für die Beispiel-Diskette.

Benutzer von Festplattensystemen
Benutzer von Festplattensystemen sollten die Blitz2 Programme und Bespiele in einem Ordner auf der Festplatte installieren. Dafür booten Sie von der Festplatte und erzeugen einen neuen Ordner mit dem Namen „Blitz2“.

Legen Sie die Blitz2 Programm-Diskette in das Laufwerk ein, klicken Sie das Symbol an und ziehen Sie die drei Symbole *TED*, *Blitz2* und *DEFLIBS* in den neuen Ordner auf der Festplatte. Alle Unterverzeichnisse, wie z.B. *TOOLS*, müssen ebefalls in den neuen Ordner auf der Festplatte kopiert werden. Dann legen Sie die Beispiel-Diskette ein und ziehen die Ordner von dieser Diskette ebenfalls in den Blitz2 Ordner, den Sie auf der Festplatte angelegt haben.

Alles was Sie jetzt noch brauchen, ist ein „ASSIGN“-Befehl, der dem Blitz2 Programm mitteilt, wo die Dateien zu finden sind. Um den Befehle einzufügen, laden Sie die Datei „s:startup-sequence“ in einen Texteditor, wie z.B. *Ted*, suchen die Stelle, an der sich andere „ASSIGN“-Befehle befinden und fügen folgende Zeile ein:

```
ASSIGN Blitz2: FestplattenName:Blitz2VerzeichnisName
```

Wenn Ihre Festplatte die Bezeichnung „Work“ trägt und Sie haben die Daten von der Blitz2 Programmdiskette in einen Ordner mit dem Namen „Blitz2“ kopiert, dann sollte die Zeile für Ihre Startup-Sequenz so aussehen

```
ASSIGN Blitz2: Work:Blitz2
```

Wenn Sie die Zeile eingefügt haben, starten Sie Ihren AMIGA erneut, um die geänderte Startup-Sequenz ausführen zu lassen

Alles Klar ?

Nachdem Sie eine funktionsfähige Sicherungskopie der Diskette angelegt oder Blitz2 auf der Festplatte installiert haben, ist es an der Zeit, Blitz2 einmal laufen zu lassen

Klicken Sie auf das Blitz2 Symbol um den Editor/Copiler zu starten. Der Editor meldet sich mit einem Copyright-Vermerk (den Sie nicht ignorieren sollten). Klicken Sie auf „OkeeDokee“ und los geht's.

Falls Blitz2 nicht richtig startet gibt es hierfür drei mögliche Gründe:

1. PLEASE INSERT VOLUME Blitz2: IN ANY DRIVE
Wenn diese Nachricht erscheint, müssen Sie Blitz2 beenden und die oben beschriebenen Schritte nochmals überprüfen. Benutzer von Floppy-Systemen sollten darauf achten, daß die Sicherungskopie, die Sie benutzen, den Namen „Blitz2“ trägt. Benutzer von Festplatten-Systemen dürfen das „Assign“ in der Startup-Sequenz nicht vergessen.

2. ES PASSIERT NICHTS
Wenn Sie das Symbol für Blitz2 in der Workbench anklicken und das Programm läuft nicht, sondern kehrt sofort zur Workbench zurück, dann liegt das daran, daß der Editor TED sich nicht im selben Verzeichnis wie das Blitz2-Programm befindet. In diesem Fall ziehen Sie das Symbol des *TED* von der Blitz2-Programmdiskette in den selben Ordner, in dem auch das Blitz2-Symbol ist.

3. PLEASE INSERT VOLUME BlitzLibs: IN ANY DRIVE
Der Grund für diese Nachricht ist der, daß sich die Datei „DefLibs“ nicht im selben Ordner wie Blitz2 und *TED* befindet. Wenn *DefLibs* nicht vorhanden ist, versucht Blitz2, alle internen Befehle vom Laufwerk „BlitzLibs:“ zu lesen (dies sei nur für fortgeschrittene Benutzer erwähnt).

Die Beispielprogramme

Wenn Sie es bis hierher ohne die oben genannten Probleme geschafft haben, können Sie einmal einige Beispiele ausprobieren.

Wählen Sie *LOAD* aus dem Menü, legen Sie die Beispiel-Diskette ein und laden Sie eines der Beispiele. Alle Dateien mit dem Zusatz *.bb2* sind Quelldateien, die in den Blitz2 Editor/Compiler geladen werden können. Sobald Sie eine .bb2-Datei geladen haben, lesen Sie den Quellcode durch und versuchen Sie herauszufinden, was das Programm wohl macht. Dann wählen Sie *Compile and Run* aus dem Menü aus.

Das einzige, was jetzt noch schiefgehen kann, ist, daß das aktuelle Verzeichnis nicht stimmt.

Das aktuelle Verzeichnis

Wenn Blitz2-Programme auf Dateien der Platte zugreifen müssen (z.B. um Graphiken zu laden), wird standardmäßig immer im aktuellen Verzeichnis danach gesucht. Darunter versteht man das Verzeichnis, von dem aus Sie das betreffende Programm gestartet haben.

Damit Blitz2 die benötigten Dateien findet, ist es notwendig, vor dem Programmstart in das Verzeichnis zu wechseln, in dem sich die Dateien befinden. Hierzu dient der mit *CD* („Change Directory“) beschriftete Knopf des File-Requesters.

Wenn Sie also den Pfadnamen im File-Requester geändert haben, um ein Beispielprogramm zu laden,

müssen Sie zunächst auf den CD-Knopf klicken, bevor Sie *OK* anwählen.

Tritt eine Fehlermeldung wie „Couldn't Load Shape“ o.ä. auf, liegt das daran, daß nicht in das aktuelle Verzeichnis gewechselt wurde. Drücken Sie *ESC*, um den Debugger zu verlassen und zum Editor zurückzukehren.

Wenn das Programm den Rechner zum Absturz bringt, dann ist die Ursache dafür der selbe Fehler, allerdings war die Fehler-Überprüfung im Menü *Compiler Options* ausgeschaltet.

Zusammenfassung

Um das Kapitel kurz zusammenzufassen:

Danke, daß Sie Blitz2 erworben haben.

Legen Sie zuallererst eine Sicherungskopie der Disketten an. Wenn Sie ein Festplattensystem besitzen, legen Sie ein Verzeichnis für Blitz2 an und kopieren Sie alle Dateien und Unterverzeichnisse von den Disketten in dieses Verzeichnis. Fügen Sie dann folgende Zeile in die Startup-Sequenz ein:

```
ASSIGN BLITZ2: FestplattenName:BlitzVerzeichnisName
```

Wenn sich alle drei Dateien (*Blitz2*, *TED* und *DefLibs*) in dem selben Verzeichnis befinden, klicken Sie auf das Blitz2-Symbol. Wählen Sie *Load* aus dem Menü und wechseln Sie zu dem Verzeichnis „Demo“ auf Diskette 2. Wählen Sie irgendeine Quelldatei, das sind alle diejenigen mit dem Zusatz *.bb2*.

Wenn Sie auf den CD-Knopf klicken, bevor Sie *OK* auswählen, können Sie sicher sein, daß das Programm, nachdem es compiliert wurde, auch alle benötigten Dateien findet. Mit dem CD-Befehl wird in das Verzeichnis gewechselt, in dem sich auch das Programm befindet.

Kapitel 2

Grundlegendes über Blitz2

- Mein erstes Programm
- Der Print-Befehl
- Formatiertes Drucken
- Einfache Variablen
- Einfache Schleifen
- Verschachtelte Schleifen
- Die Benutzung von String-Variablen
- Der Programmablauf
- Sprungbefehle
- Benutzereingaben
- Felder

Das folgende Kapitel wendet sich an diejenigen Benutzer, die noch keine Programmiererfahrung in BASIC haben. Es liefert eine schrittweise Einführung in die grundlegenden Elemente von Blitz2.

Sobald Blitz2 betriebsbereit ist (s. vorhergehendes Kapitel), betätigen Sie den „OKEE DOKEE“-Knopf und Sie können Ihr erstes Programm schreiben.

Wenn Sie Schwierigkeiten mit dem Blitz2-Editor haben sollten, sehen Sie im Anhang 1 nach. Dort ist der Editor Ted ausführlich erklärt.

Mein erstes Programm

Geben Sie die folgenden zwei Befehle ein:

```
PRINT "Dies ist mein erstes Blitz2 Programm!"
MouseWait
```

Dann wählen Sie *COMPILE&RUN* im oberen rechten Menü.
Wenn Sie das Programm korrekt eingegeben haben, erhalten Sie ein CLI-Fenster mit Ihrer Mitteilung. Betätigen Sie den Mausknopf und Sie sind wieder im Editor.

Das ist schon alles.

Der Print-Befehl

Positionieren Sie den Cursor auf das Wort `Print` in Ihrem Program und drücken Sie die Help-Taste. Die Syntax für den Print-Befehl erscheint dann oben auf dem Bildschirm. Dort steht jetzt:

```
Print Expression[,Expression...]
```

Die eckigen Klammern sagen aus, daß der Print-Befehl mit beliebig vielen Ausdrücken versorgt werden kann. Die Ausdrücke müssen durch Kommata getrennt werden. Ein Ausdruck kann eine beliebige Zahl, eine

Zeichenkette (ein Text in „Anführungszeichen“), eine Variable oder eine BASIC-Formel sein. Das folgende Beispiel enthält alle diese Möglichkeiten.

Vergessen Sie den MouseWait-Befehl nicht wenn Sie dieses Programm testen, denn sonst druckt Blitz2 die Meldung und kehrt sofort zum Editor zurück, bevor Sie Zeit hatten, sie zu lesen.

```
Print 3,"Autos",a,a*7+3
```

Die folgende Zeile sollte in dem CLI-Fenster erscheinen.

```
3Autos03
```

Wenn Sie Zwischenräume zwischen die verschiedenen Ausdrucke einfügen, wie z.B. so:

```
Print 3," Autos",a," ",a*7+3
```

dann erhalten Sie folgende Zeile

```
3 Autos 0 3
```

Formatiertes Drucken

Der Format-Befehl ermöglicht es, Zahlen in unterschiedlichen Formaten auszudrucken. Dies ist z.B sinnvoll wenn Sie Zahlen in Spalten ausgeben wollen.

Der NPrint-Befehl dient dazu, den Cursor nach der Ausgabe automatisch in die nächste Zeile zu bewegen.

```
Format "###.00"
Nprint 23,5
Nprint 10
Nprint ,5
Nprint 0
MouseWait
```

Einfache Variablen

Was eine Programmiersprache mächtig macht, ist ihre Fähigkeit, Zahlen und Text zu verändern. Um diese Informationen zu speichern werden Variablen verwendet.

Der folgende Befehl weist der Variablen `a` den Wert `5` zu :

```
a=5
```

Jetzt besitzt die Variable `a` den Wert `5`. Man kann den Computer jetzt anweisen, `1` zu dem Wert von `a` zu addieren, sodaß das Ergebnis `6` ist :

```
a=a+1
```

Ein Ausdruck kann auch mehrere Rechenoperationen enthalten. Um zu erreichen, daß eine bestimmte Operation vor der anderen ausgeführt wird, wird diese in Klammern eingeschlossen:

```
a=(a+3)*7
```

An dieser Stelle sei auf Anhang 5 verwiesen, der eine Liste aller Operatoren und deren Präzedenz (die Reihenfolge, in der die Operationen ausgeführt werden) enthält.

In dem folgenden Programm wird `a` zunächst der Wert `0` zugewiesen, dann viermal `12` addiert und der Wert ausgedruckt.

```
a=0
a=a+12:Nprint a
a=a+12:Nprint a
a=a+12:Nprint a
a=a+12:Nprint a
MouseWait
```

Wie Sie sehen, können zwei Befehle in der selben Zeile stehen wenn sie durch einen Doppelpunkt getrennt werden. Im Übrigen ist der erste Befehl `a=0` nicht nötig, da allen Variablen in Blitz2 grundsätzlich zunächst der Wert 0 zugewiesen wird.

Einfache Schleifen

Das folgende Programm druckt das 1 mal 12. Anstatt 12 Zeilen einzugeben, wird eine For...Next-Schleife verwendet. Durch eine Schleife wird das Programm angewiesen, einen Programmabschnitt mehrmals zu wiederholen.

Durch `For i=0 To 12..Next` wird erreicht, daß die Befehle zwischen `For` und `Next` 12 mal ausgeführt werden. Dabei wird die Variable `i` als Zähler verwendet.

Der Stern `*` ist der Multiplikations-Operator; `a=i*12` bedeutet, daß die variable `a` anschließend den 12-fachen Wert der Variablen `i` besitzt. Da `i` aber von 1 bis 12 hochgezählt wird, erhält `a` die Werte 12, 24, 36 usw.

```
For i=1 To 12
      a=i*12
      Nprint i, "*",12,"=",a
Next
MouseWait
```

Beachten Sie, daß die beiden Zeilen innerhalb der Schleife eingerückt sind. Dies dient dazu, das Programm übersichtlicher zu machen, sodaß man leichter erkennen kann, welche Abschnitte innerhalb von Schleifen sind und welche nicht.

Um Zeilen einzurücken wird die Tabulator-Taste verwendet.

Wenn Sie nun die erste Zeile abändern, sodaß dort jetzt `For i=1 To 100` steht, werden Sie sehen, daß der Rechner auch kein Problem hat, dies zu berechnen.

Sie können auch die Zahl 12 in den ersten 3 Zeilen abändern um das Einmaleins für andere Zahlen zu erstellen.

Verschachtelte Schleifen

Das folgende Programm ist ein Beispiel für die Verschachtelung von Schleifen. Verschachtelte Schleifen sind

Schleifen die innerhalb anderer Schleifen sind. Dabei werden die Anweisungen der inneren Schleife weiter eingerückt, sodaß sich leichter erkennen läßt, ob für jeden For-Befehl auch ein entsprechender Next-Befehl vorhanden ist.

```
For y=1 To 12
      For x=1 To 12
            NPrint y,"*",x,"=",x*y
      Next
Next
MouseWait
```

Die Verschachtelung der `For x=1 To 12` Schleife innerhalb der `For y=1 To 12` Schleife hat zur Folge, daß die Zeile `NPrint...` 12*12mal ausgeführt und jedesmal eine neue Kombination von `x` und `y` berechnet wird.

While...Wend und Repeat...Until
Außer `For...Next` gibt es noch zwei weitere einfache Methoden in Blitz2, um Schleifen zu realisieren.

`While...Wend und Repeat...Until` Schleifen werden wie folgt geschrieben:

```
While a<20
      Nprint a
      a=a+1
Wend

Repeat
      Nprint a
      a=a+1
Until a>=20
```

Wie viele BASIC-Befehle sind auch diese einigermaßen selbsterklärend: das Innere der `While...Wend`-Schleife wird wiederholt, SOLANGE die Bedingung erfüllt ist, die `Repeat...Until`-Schleife wird durchlaufen, BIS die Bedingung erfüllt ist.

Die Bedingung die angegeben wird, kann ein beliebiger Ausdruck sein wie `While a+10<50`, `While f=0`, `While b<>x*2`, usw.

Der Unterschied zwischen diesen beiden Schleifen ist, daß bei einem Anfangswert von `a` größer als `20` die `Repeat...Until`-Schleife einmal durchlaufen wird, die `While...Wend`-Schleife jedoch gar nicht (die `While`-Schleife wird daher auch abweisende Schleife genannt).

Endlosschleifen
Wenn ein Programm in einen Zustand gerät, in dem eine Schleife nicht mehr verlassen wird, spricht man von einer Endlosschleife. In einer solchen Situation muß der Programmierer in der Lage sein, das Programm von außen zu beenden.

Ein Programm kann mit der *Ctrl/Alt-C* Kombination unterbrochen werden. Hierzu müssen die *Ctrl*-Taste und die linke *Alt*-Taste gleichzeitig gedrückt und gehalten werden und zusätzlich das *C* gedrückt werden. Das Programm wird dann angehalten und der Debugger erscheint auf dem Bildschirm. Um den Debugger zu verlassen und zum Editor zurückzukehren, drücken Sie die *Esc*-Taste (links oben auf der Tastatur). Der Debugger wird in Kapitel 6 behandelt.

Die Benutzung von String-Variablen

Variablen, die keine Zahlenwerte sondern Buchstaben (Text) enthalten werden Zeichenketten oder String-Variablen genannt. Um String-Variablen zu kennzeichnen, müssen sie ein $-Zeichen hinter dem Namen erhalten. Das folgende Beispiel zeigt eine einfache Anwendung von Strings:

```
a$="Simon"
Nprint a$
MouseWait
```

Wie bei den numerischen Variablen wird auch bei Strings das „=“-Zeichen dazu verwendet, der Variablen einen Wert zuzuweisen. Das „+“-Zeichen dient dazu, String-Variablen zu addieren, also aneinander zu fügen:

```
a$="Simon":b$="Armstrong":c$=a$+b$
```

Die Variable `c$` enthält anschließend die Zeichenkette „Simon Armstrong“. Funktionen, die mit Strings arbeiten sind in Kapitel 6 des „Blitz Referenzhandbuchs“ aufgeführt.

Der Programmablauf

Vielfach muß ein Programm bei der Ausführung entscheiden, ob der eine oder der andere Befehl auszuführen ist; dies nennt man den Programmablauf. Die `If..Then`-Anweisung dient dazu, den nachfolgenden Befehl nur dann ausführen zu lassen, wenn eine bestimmte Bedingung erfüllt ist. Im folgenden Beispiel wird nur dann das Wort „Hallo“ ausgegeben, wenn die Variable `a` den Wert 5 hat:

```
If a=5 Then Print "Hallo"
```

Unter Verwendung des `If...EndIf`-Befehls kann das Beispiel so verändert werden, daß ein ganzer Programmabschnitt nur dann ausgeführt wird, wenn `a` gleich 5 ist:

```
If a=5
      Print "Hallo"
      a=a-1
EndIf
```

Mit dem `Else`-Befehl kann alternativ ein anderer Programmabschnitt ausgeführt werden, wenn die Bedingung nicht erfüllt ist:

```
If a=5
      Print "Hallo"
Else
      Print "Auf Wiedersehen"
EndIf
```

Der Programmcode wird widerum innerhalb der Strukturblöcke eingerückt um die Lesbarkeit zu erhöhen, genau wie bei den Schleifen.

Die Bedingung, die in dem `If`-Befehl genannt ist, kann ein beliebig komplizierter Ausdruck sein, wie die folgenden Beispiele zeigen:

```
If a=1 or b=2
If a>b+5
If (a+10)*50<>b/7-3
```

Anhang 5, am Ende dieses Handbuchs, enthält eine vollständige Beschreibung des Gebrauchs von Operatoren und deren Präzedenz.

Sprungbefehle

Oftmals ist es nötig, innerhalb des Programms zu einem anderen Abschnitt zu „springen“. Hierfür dienen die Befehle `Goto` und `Gosub`.

Die Stelle, zu der das Programm springen soll muß eine Sprungmarke (Label) besitzen, sodaß angegeben werden kann, zu welcher Stelle gesprungen werden soll.

Im folgenden wird die Sprungmarke „start“ verwendet:

```
Goto start
Nprint "Hallo"
start
MouseWait
```

Da durch den `Goto`-Befehl das Programm dazu veranlasst wird, zur Marke „start“ zu springen, wird der Text „Hallo“ nie ausgegeben.

Der Befehl `GoSub` wird dazu benutzt, um zu einem Unterprogramm (Subroutine) zu springen. Ein Unterprogramm ist ein Programmabschnitt, der mit einem `Return`-Befehl abgeschlossen wird. Nachdem die Anweisungen des Unterprogramms ausgeführt wurden, wird durch den `Return`-Befehl wieder zu der Zeile mit dem `GoSub`-Kommando zurückgesprungen und dort fortgefahren.

```
.start:
GoSub message
GoSub message
GoSub message
MouseWait
End
.message
NPrint "Hallo"
Return
```

Beachten Sie, daß den Sprungmarken ein Punkt vorangestellten ist. Dadurch werden sie vom Editor als solche erkannt und erscheinen in einer Liste auf der rechten Bildschirmseite. Wenn Sie mit der Maus in diese Liste klicken, springt der Cursor automatisch zu der Marke im Programm. Beim Erstellen großer Programme ist dies sehr nützlich.

Benutzereingaben

Vielfach muß ein Programm auf Eingaben vom Benutzer warten, entweder durch die Tastatur oder durch die Maus. Der `MouseWait`-Befehl, zum Beispiel, hält das Programm an, bis der Benutzer den linken Mausknopf betätigt.

Die Befehle `Edit` und `Edit$` erwarten Eingaben von der Tastatur, ähnlich wie der `Input`-Befehl in anderen Programmiersprachen auch.

Das folgende Beispiel fragt den Benutzer nach seinem Namen und speichert die Eingabe in einer String-Variablen:

```
Print "Ihr Name bitte ?"
a$=Edit$(80)
NPrint "Hallo",a$
MouseWait
```

Die Zahl 80 in dem Befehl `Edit$(80)` bezieht sich auf die Anzahl Zeichen, die der Benutzer eingeben kann.

Um Zahlen einzulesen, wird die Funktion `Edit` verwendet. Der Befehl `a=Edit(80)` erwartet die Eingabe einer maximal 80stelligen Zahl und speichert diese in der Variablen `a`.

Felder

Programme müssen häufig Gruppen von Zahlen oder Strings verarbeiten. Variablen, die solche Gruppen darstellen, werden Felder (Arrays) genannt. Wenn z.B. eine Gruppe von zehn Zahlen verarbeitet werden sollen, die alle in einer gewissen Beziehung zueinander stehen, so kann statt zehn einzelner Variablen auch ein Feld definiert werden.

Hierzu dient die Anweisung `Dim`:

```
Dim a(10)
```

Die Variable `a` kann nun 10 Zahlen aufnehmen (eigentlich 11, siehe Kapitel 3). Um auf die einzelnen Werte zuzugreifen, wird der Index des Feldelements in der Klammer hinter dem Variablennamen angegeben:

```
a(1)=5
a(2)=6
a(9)=22
NPrint a(9)
a(1)=a(1)+a(2)
Nprint a(1)
```

Das wichtige dabei ist, daß der Index, der das Feldelement angibt, wiederum eine Variable sein kann. Wenn `i=2` ist, dann bezieht sich der Ausdruck `a(i)` auf die selbe Variable wie `a(2)`.

Im folgenden werden 5 Strings in einer `For...Next`-Schleife vom Benutzer eingelesen. Da die Strings in einem Feld gespeichert werden, können sie wiederum in einer Schleife ausgedruckt werden:

```
Dim a$(20)
NPrint "Geben Sie 5 Namen ein"
For i=1 To 5
      a$(i)=Edit$(80)
Next
NPrint "Die eingegebenen Namen sind"
For i=1 To 5
      NPrint a$(i)
Next
MouseWait
```

Kapitel 3

Datentypen, Felder und Listen

- Numerische Datentypen
- Der Default-Datentyp
- Die Data-Anweisung
- Wertüberlauf
- Der Datentyp String
- Sytemkonstanten
- Die NewType-Anweisung
- Felder innerhalb von NewTypes
- Die UsePath-Anweisung
- Der Datentyp Pointer
- Felder
- Listen

Numerische Datentypen

Blitz2 kennt 6 verschiedene Variablentypen. 5 davon sind numerische Datentypen, die dazu dienen, Zahlenwerte mit unterschiedlichen Wertebereichen und unterschiedlicher Genauigkeit zu speichern, der sechste Datentyp speichert Folgen von Buchstaben, also Strings (Text).

Die untenstehende Tabelle gibt den Wertebereich, die Genauigkeit und den Speicherbedarf für alle numerischen Datentypen an:

Typ	**Zusatz**	**Bereich**	**Genauigkeit**	**Bytes**
Byte	.b	+/- 128	ganzzahlig	1
Word	.w	+/- 32768	ganzzahlig	2
Long	.l	+/- 2147483648	ganzzahlig	4
Quick	.q	+/- 32768.0000	1/65536	2
Float	.f	+/- 9*10^18	1/10^184	4

Der Datentyp „Quick“ ist eine Festkomma-Größe mit geringerer Genauigkeit als Fließkommatypen, dafür aber schneller.

„Float“ ist eine Fließkomma-Größe, die von der „Amiga Fast Floating Point Bibliothek“ unterstützt wird.

Bei der Vereinbarung einer Variablen wird deren Datentyp durch den entsprechenden Zusatz (Suffix) bestimmt. Damit ist der Datentyp festgelegt und braucht bei allen weiteren Zugriffen nicht mehr angegeben zu werden, außer bei String-Variablen.

Es folgen einige Beispiele für numerische Variablen und deren Zusatz.

```
mychar.b=127
my_score.w=32000
chip.l=$dff000    ;$ bezeichnet einen Hexadezimal-Wert
speed3.q=500/7    ;Quick-Variablen haben eine Genauigkeit von 3 Dezimalpunkten
light_speed.f=3e8 ;e ist der Exponent, d.h. 3x10^8
```

Der Default-Datentyp

Wird bei der Vereinbarung einer Variablen kein Zusatz angegeben, so wird automatisch der Default-Datentyp (Vorbesetzung) zugewiesen. Ist nichts weiter angegeben, so ist dies „Quick“.

Um die Vorbesetzung zu ändern, benutzt man den Befehl `DEFTYPE`. Alle Variablen, die anschließend ohne ausdrückliche Angabe des Datentyps vereinbart werden, sind vom neuen Default-Typ.

Folgt dem Befehl `DEFTYPE` unmittelbar eine Liste von Variablennamen, dient er dazu, den Datentyp für diese Variablen zu bestimmen, die Vorbesetzung jedoch unverändert zu lassen. Der Befehl bekommt dadurch also eine andere Bedeutung.

Das nachfolgende Beispiel erläutert den Gebrauch der beiden Varianten von `DEFTYPE`.

```
a=20              ;a ist vom Typ Quick
DEFTYPE .f        ;Der neue Default-Typ ist jetzt Float
b=20              ;b ist vom Typ Float
DEFTYPE .w c,d    ;c & d sind vom Typ Word, Default ist immer noch Float
```

Anmerkung: die erste Form von `DEFTYPE` kann mit „ändere Default-Typ“ übersetzt werden, die zweite Form bedeutet „Definiere Typ“.

Als Default-Typ kann auch ein `NewType` festgelegt werden, näheres dazu im nächsten Abschnitt.

Es gibt noch weitere Blitz2-Strukturen, wie `Data`, `Peek`, `Poke`, sowie Funktionen, die ebenfalls den Default-Datentyp verwenden, sofern kein Zusatz angegeben ist.

Die Data-Anweisung

Die `Data`-Anweisung dient dazu, einer Reihe von Variablen eine Liste von Werten zuzuordnen. Die `Restore`-Anweisung dient dazu, den Datenzeiger auf eine bestimmte Data-Anweisung zeigen zu lassen.

Um zu bestimmen, von welchen Datentyp die Werteliste ist, wird der entsprechende Zusatz *.typ* angefügt.

Das folgende Beispiel erläutert den Gebrauch von `Data` in Blitz2:

```
Read a,b,c
Restore myfloats
Read d.f
Restore mystrings
Read e$,f$,g$
myquicks:
      Data 20,30,40
myfloats:
      Data .f20.345,10.7,90.111
mystrings:
      Data$ "Guten","Morgen","Simon"
```

Anmerkung: wenn der Datenzeiger auf einen anderen Datentyp zeigt als die Variable, die in der `Read`-Anweisung aufgeführt ist, tritt der Compilerfehler „Mismatched Types“ („Nicht-gleichwertige Datentypen“) auf.

Wertüberlauf & vorzeichenlose Zahlen

Wird einer Variablen ein Wert zugewiesen, der außerhalb des Wertebereichs des Datentyps liegt (also zu groß ist), tritt ein Überlauf-Fehler auf. Das folgende Beispiel erzeugt einen solchen Überlauf-Fehler, wenn es ausgeführt wird:

```
a.w=32767   ;a ist ein Word mit dem Wert 32767
a=a+1       ;Überlauf, der Wert ist zu groß
```

Die Überprüfung von Überlauf-Fehlern kann in der Compiler-Konfiguration unter „Run Time Errors Options“ ein- oder ausgeschaltet werden. Die Voreinstellung ist „aus“, sodaß das obige Beispiel keine Fehlermeldung erzeugen würde.

Es kann vorkommen, daß eine ganzzahlige Variable (Integer) vorzeichenlos sein soll, also nur positive Werte annehmen kann. Byte-Variablen sollen z.B. häufig einen Wertebereich von 0 bis 255 haben statt -128 bis +127. Um dies zu ermöglichen, muß die Überprüfung von Überlauf-Fehlern in der Compiler-Option „Error Checking“ ausgeschaltet werden

Der Datentyp String

Ein String ist eine Variable, die dazu dient, eine Zeichenkette zu speichern, also Text. Eine String-Variable wird mit dem Zusatz *.s* oder dem traditionellen *$*-Zeichen gekennzeichnet.

Anders als bei numerischen Variablen, muß dieser Zusatz grundsätzlich immer angegeben werden. Dies hängt damit zusammen, daß der Name einer String-Variablen gleichzeitig auch für eine numerische Variable verwendet werden kann.

Das folgende Beispiel ist also zulässig:

```
a$="Hallo"
a.w=20
NPrint a,a$
```

Systemkonstanten

Unter Systemkonstanten werden spezielle Variablen verstanden, die von Blitz2 dazu verwendet werden, bestimmte unveränderliche Werte zu speichern. Die folgenden Variablen-Namen sind also reserviert für die angeführten Werte:

```
Pi          3.1415
On          -1
Off         0
True        -1
False       0
```

Zusammenfassung der einfachen Datentypen
Blitz2 kennt 6 einfache Datentypen, die Typen *Byte*, *Word* und *Long* sind vorzeichenbehaftete 8-, 16- und 32-Bit-Größen.

Der Datentyp *Quick* ist eine Festkomma-Größe, die nicht so genau ist wie Fließkomma-Größen, aber schneller.

Der Datentyp *Float* ist die von der „Amiga Fast Floating Bibliothek“ unterstützte Fließkomma-Größe.

String ist die standardmäßige BASIC-Implementation für Zeichenketten.

Mit Hilfe der `DefType`-Anweisung, können Variablen eines bestimmten Typs vereinbart werden, ohne daß jeweils der Zusatz angegeben werden muß.

Ist eine Variable einmal vereinbart, kann der Zusatz für den Datentyp weggelassen werden, außer bei *String*-Variablen. Dort muß er grundsätzlich immer angegeben werden.

Der Datentyp einer Variable ist innerhalb eines Programms unveränderlich und es kann immer nur eine Variable eines Namens vereinbart werden. Eine Ausnahme bilden wiederum die Strings, bei denen der gleiche Name auch für eine numerische Variable vergeben werden kann.

Die NewType - Anweisung

Zusätzlich zu den 6 einfachen Datentypen, die in Blitz2 zur Verfügung stehen, haben Programmierer die Möglichkeit, eigene Datentypen zu entwerfen.

Der Befehl `NewType` ermöglicht es,verschiedene Datentypen,die inhaltlich zusammengehören, zu einem neuen Datentyp zusammenzufügen, ähnlich einem Datensatz in einer Datenbank oder einer Struktur in C. Das folgende Beispiel zeigt, wie Variablen, die den Namen, das Alter und die Größe einer Person enthalten, einem neuen Variablentyp zugeordnet werden.

```
NEWTYPE .Person
      name$
      alter.b
      groesse.q
End NEWTYPE

a.Person\name = "Harry",20,1.8
NPrint a\height
```

Ist ein `NewType` einmal definiert, können Variablen dieses Typs vereinbart werden, indem der Zusatz *.NeuerTypname* angefügt wird, z.B.: `a.Person`.

Auf einzelne Elemente innerhalb eines `NewType` kann durch das Zeichen „\“ (Backslash) zugegriffen werden, z.B.: `a\groesse=a\groesse+1`.

Bei der Definition eines `NewType` erhalten Variablen, für die kein Suffix angegeben ist den Datentyp des vohergehenden Elementes. Es können auch mehrere Elemente in einer Zeile aufgeführt werden, wenn sie durch Doppelpunkte getrennt sind. Es folgt ein weiteres Beispiel eines `NewType`:

```
NewType .nme
      x.w:y:z     ;y & z sind auch von Typ Word (s.o.)
      wert.w
      speed.q
      name$
End NewType
```

Der Zugriff auf Strings innerhalb eines `NewType` benötigt keinen *$* oder *.s*-Zusatz wie bei normalen Zeichenketten. Ist ein solcher Suffix angegeben, führt dies zur Compilermeldung „Garbage at End of Line“

(„Schrott am Ende der Zeile“). Bezogen auf das erste Beispiel heißt das:

```
a\name="Jimi Hendrix"   ;richtig
a\name$="Bob Dylan"     ;falsch
```

Zuvor definierte NewTypes können in darauffolgenden `NewType`-Definitionen weiterverwendet werden. Es folgt ein Beispiel für eine `NewType`-Definition, die einen anderen `NewType` enthält:

```
NewType .vektor
      x.q
      y.q
      z.q
End NewType

NewType .object
      position.vektor
      geschwindigkeit.vektor
      beschleunigung.vektor
End NewType

DefType .object meinschiff    ;s. nächster Absatz
meinschiff\position\x=100,0,0
```

Beachten Sie, daß jetzt zwei Backslashes nötig sind um auf Elemente des `NewType` „meinschiff“ zuzugreifen. Das ganze sieht jetzt aus wie ein Pfadname unter DOS.

Ein einmal definierter `NewType` kann im Zusammenhang mit beiden Formen des `DefType`-Befehles verwendet werden, genauso wie jeder andere Datentyp.

Felder innerhalb von NewTypes

Außer einfachen Datentypen und anderen NewTypes können auch Felder als Bestandteile für NewTypes verwendet werden. Zu ihrer Kennzeichnung werden die eckigen Klammern [und] verwendet.

Im Gegensatz zu normalen Feldern sind Felder innerhalb von NewTypes begrenzt auf eine Dimension und ihre Größe muß durch eine Konstante definiert werden, nicht durch eine Variable. Ein Feld kann einen beliebigen Datentyp besitzen, auch Felder vom Typ eines `NewType` sind zulässig.

Ein weiterer Unterschied zu herkömmlichen Feldern besteht darin, daß die Angabe in den eckigen Klammern genau die Größe des Feldes angibt. (Bei herkömmlichen Feldern ist dies nicht der Fall, siehe nächster Abschnitt) Die Anweisung `adresse.s[4]` reserviert Speicherplatz für 4 Strings mit den Indizes 0 bis 3.

Das folgende Beispiel erläutert den Gebrauch von Feldern in NewTypes:

```
NewType .datensatz
      name$
      alter.w
      adresse.s[4]      ; dasselbe wie adresse$[4]
End NewType

DefType .datensatz p
p\adresse[0]="10 St Kevins Arcade"
p\adresse[1]="Karangahape Road"
```

```
p\adresse[2]="Auckland"
p\adresse[3]="New Zealand"
For i=0 TO 3
      NPrint p\adresse[i]
Next
MouseWait
```

Wenn kein [index] angegeben ist, wird das erste Feldelement (0) verwendet.

Wird ein Feld mit der Länge 0 innerhalb eines `NewType` vereinbart, so bildet dies eine „Union“ mit dem nachfolgenden Element, d.h. beide Elemente belegen den selben Speicherplatz.

Die UsePath-Anweisung

Bei der Verwendung von verschachtelten NewTypes können die Pfadnamen für den Zugriff auf Elemente innerhalb von Elementen sehr lang werden.

Wenn eine Routine nur auf ein Element zugreift, können lange Pfadnamen durch die Verwendung von `UsePath` vermieden werden. Wird auf ein Feld zugegriffen, das nicht mit dem Namen der Wurzel-Variablen beginnt, sondern mit einem Backslash, fügt der Compiler den Namen der Wurzel-Variablen ein, der in der `UsePath`-Anweisung definiert wurde.

Im folgenden Beispiel

```
UsePath Shapes (i)\pos
For i=0 To 9
      \x+10
      \y+20
      \z-10
Next
```

wird der Variablenname vom Compiler erweitert, sodaß sich folgendes ergibt:

```
For i=0 To 9
      shapes(i)\pos\x+10
      shapes(i)\pos\y+20
      shapes(i)\pos\z-10
Next
```

Durch die `UsePath`-Anweisung wird das Programm erheblich besser lesbar und es wird Schreibarbeit gespart.

Es sei darauf hingewiesen, daß `UsePath` eine Compiler-Direktive ist, d.h. die Anweisung wird bereits vom Compiler ausgewertet und nicht erst vom Prozessor zur Laufzeit. Dies hat Auswirkungen, wenn in dem Programm `UsePath`-Anweisungen übersprungen werden: der jeweils gültige `UsePath` ist trotzdem derjenige, der im Quellcode vor der entsprechenden Anweisung steht.

Felder

Felder in Blitz2 entsprechen den normalen BASIC-Konventionen. Alle Felder müssen dimensioniert werden, bevor sie verwendet werden können. Sie können aus beliebigen Datentypen bestehen (einfache oder

`NewType`) und sie können beliebig viele Dimensionen besitzen.

Alle Felder besitzen einen Index von 0 bis n, wobei n die Größe des Feldes ist. Wie in anderen BASIC-Versionen auch, kann ein Feld eigentlich ein Element mehr als n aufnehmen, es wird nämlich von 0 bis n einschließlich gezählt. Das Feld a(50) besitzt also 51 Elemente, von 0 bis 50.

Bei der Vereinbarung eines Feldes wird, wie bei allen anderen Variablen auch, der Default-Datentyp angenommen, wenn nicht ausdrücklich ein *.typ*-Zusatz angegeben ist:

```
Dim a.w(50) ;Ein Feld vom Typ Word
```

Durch die Möglichkeit, Felder aus `NewType`-Datentypen zu vereinbaren, verringert sich die Anzahl von Feldern, die ein BASIC-Programm benötigt.

Betrachten wir folgendes Beispiel:

```
Dim Alienflags(100),Alienx(100),Alieny(100)
```

Unter Verwendung von `NewType`-Datentypen könnte dasselbe auch erreicht werden durch:

```
NewType .Alien
      flags.w
      x.w
      y.w
End NewType

Dim Aliens.Alien(100)
```

Es wird jetzt nur ein Feld benötigt, um auf alle benötigten „Alien“-Daten zugreifen zu können. Sollen alle x- und y-Werte von `Alien` auf einmal mit Zufallszahlen belegt werden, so kann dies folgendermaßen geschehen:

```
For k=1 To 100
      Alien(k)\x=Rnd(320),Rnd(200)
Next
```

Sollen jetzt weitere Informationen über Aliens hinzugefügt werden, müssen diese lediglich in der Definition des `NewType` eingetragen werden, es brauchen keine neuen Felder angelegt zu werden.

Anmerkung: Im Gegensatz zu den meisten anderen BASIC-Compilern ERLAUBT Blitz2 die Vereinbarung von Feldern mit variablen Dimensionen, also `Dim a(n)`. Außerdem muß für Strings in Feldern KEINE maximalen Länge angegeben werden, wie in einigen anderen Programmiersprachen.

Listen

In Blitz2 gibt es auch eine weiterentwickelte Form der Felder, die Listen. Listen sind grundsätzlich Felder, haben aber einige spezielle Eigenschaften.

Häufig wird jeweils nur ein bestimmter Abschnitt eines Feldes belegt und es wird eine gesonderte Variable mitgeführt, die Auskunft darüber gibt, wieviele Elemente des Feldes zur Zeit belegt sind. In einem solchen Fall sollte das Feld durch eine Liste ersetzt werden, die die Verwaltung erheblich vereinfacht und beschleunigt.

Dimensionierung einer Liste

Die Dimensionierung einer Liste erfolgt genau wie bei einem Feld, es wird lediglich das Wort List hinter `Dim` eingefügt. Listen sind z.Zt. auf eine Dimension begrenzt.

Hier ist ein Beispiel für die Definition einer Liste:

```
NewType .Alien
      flags.w x y
End NewType

Dim List Aliens.Alien(100)
```

Der Unterschied zu einem einfachen Feld besteht nun darin, daß Blitz2 einen internen Zähler über die Anzahl der Listenelemente sowie einen Zeiger auf das jeweils aktuelle Element mitführt. Beide werden durch die `Dim List`-Anweisung auf 0 gesetzt.

Elemente zu einer Liste hinzufügen
Zu Beginn ist jede Liste leer, Listenelemente können durch die Funktionen `AddItem` und `AddLast` hinzugefügt werden. Da Listen irgendwann einmal voll sind, liefern diese beiden Funktionen den Wert *Wahr* oder *Falsch* zurück, je nachdem, ob das Element eingefügt werden konnte oder nicht.

Im folgenden Beispiel wird ein Alien zu der zuvor dimensionierten Liste hinzugefügt:

```
If AddItem(Aliens())
      Aliens()\x=Rnd(320),Rnd(200)
EndIf
```

In beiden Aufrufen von `Aliens()` wurde kein Index in den Klammern angegeben. Obwohl Blitz2 keine Fehlermeldung ausgeben würde, wenn ein solcher Index vorhanden wäre, sollten Listen NIEMALS mit einem Index aufgerufen werden. Die leeren Klammern verweisen auf das aktuelle Listenelement, in diesem Fall das gerade hinzugefügte.

Da `AddItems` den Wert Falsch zurückliefert, wenn die Liste voll ist, kann eine `While...Wend`-Schleife verwendet werden, um die gesamte Liste aufzufüllen:

```
While AddItem(Aliens())
      Aliens()\x=Rnd(320)
      Aliens()\y=Rnd(200)
Wend
```

Die Schleife wird solange durchlaufen, bis die Liste gefüllt ist. Wenn man eine feste Zahl (z.B. 20) von Einträgen in die Liste einfügen möchte, könnte man auch eine `For...Next`-Schleife verwenden, müßte aber trotzdem nach jedem Einfügen den Zustand der Liste prüfen:

```
For i=1 To 20
      If AddItems(Aliens())
            Aliens()\x=Rnd(320)
            Aliens()\y=Rnd(200)
      EndIf
Next
```

Listen können natürlich für jeden Datentyp angelegt werden, nicht nur für Aliens, d.h. sie sind nicht nur für Spiele zu gebrauchen.

Listenverarbeitung
Wie schon erwähnt, wird das Listenelement, das zuletzt eingefügt wurde, zum aktuellen Element. Auf dieses aktuelle Element kann zugegriffen werden, indem der Listenname mit leeren Klammern () angegeben wird.

Um die Liste von vorne bis hinten zu verarbeiten, wird zunächst der interne Listenzeiger mit `ResetList` zurückgesetzt, und dann wird in einer Schleife mit Hilfe von `NextItem` schrittweise von einem Eintrag zum nächsten gegangen. Der interne Zeiger zeigt dann jeweils auf das aktuelle Element.

Im folgenden werden alle Aliens in der Liste auf eine eher ineffektive Weise bewegt (wohl zur Bildschirmmitte):

```
USEPATH Alien()
ResetList Aliens()
While NextItem(Aliens())
      If \x>160 Then \x-1 Else \x+1
      If \y>100 Then \y-1 Els \y+1
Wend
```

Die `While...Wend`-Schleife wird solange durchlaufen, bis alle Listenelemente einmal das aktuelle Element waren und bearbeitet wurden, unabhängig davon, wieviele Elemente zuvor eingefügt wurden.

Da `NextItem` den Wert *Falsch* zurückliefert, wenn kein Listenelement mehr vorhanden ist, wird an dieser Stelle die Schleife abgebrochen. Das Beispiel macht klar, wieviel bequemer und sauberer es ist, Listen zu benutzen statt herkömmlicher Felder. Statt der `For i=1 To n`-Schleife wird einfach eine `While...Wend`-Struktur verwendet, und die Listenverwaltung ist obendrein auch noch schneller.

Listenelemente löschen
Oftmals ist es notwendig, ein Element aus der Liste zu entfernen, nachdem es bearbeitet wurde. Hierzu dient die Funktion `KillItem`. In diesem Beispiel wird wiederum die Alien-Liste verwendet:

```
ResetList Aliens()
While NextItem(Aliens())
      If Aliens()\flags=-1    ; wenn flag = -1
            KillItem Aliens() ; Element löschen
      EndIf
Wend
```

Nach dem Aufruf von `KillItem` zeigt der Listenzeiger auf das vorhergehende Element und macht es damit zum aktuelle Element. Dadurch wird in der Schleife kein Element übersprungen, nachdem eines gelöscht wurde.

Listenstrukturen
Es ist zwar möglich, auf Listenelemente wie auf herkömmliche Feldelemente durch Angabe des Index zuzugreifen, dies sollte aber niemals gemacht werden.

Die logische Reihenfolge der Elemente in der Liste ist nicht notwendigerweise die selbe wie die Reihenfolge, in der die Elemente im Speicher abgelegt sind. Intern besitzt jedes Listenelement einen Zeiger auf den Vorgänger und den Nachfolger. Wenn Blitz2 auf das nächste Element zugreifen will, richtet es sich nur nach dem Zeiger auf den Nachfolger; an welcher Speicherstelle dieser sich befindet, ist völlig unerheblich.

Beim Hinzufügen eines Elementes wird dieses an einer beliebigen Stelle im Speicher untergebracht. Seine Speicheradresse wird im Nachfolger-Zeiger des vorhergehenden Elementes hinterlegt und dessen alter Wert wird in den Nachfolger-Zeiger des neuen Elementes übernommen.

Nicht so recht verstanden? Macht auch nichts, greifen Sie einfach nie auf Listenelemente über den Feldindex zu!

Der Datentyp Pointer
Der Datentyp *Pointer* (Zeiger) in Blitz2 ist eine komplizierte Angelegenheit. Wenn eine Variable als Zeiger definiert wird, wird gleichzeitig festgelegt, auf welchen Datentyp sie zeigt. Im folgenden Beispiel wird `groesster` als Zeiger auf den Typ `.Kunde` definiert.

```
DefType *groesster.Kunde
```

Die Variable `groesster` ist einfach ein Doppelwort, das die Speicheradresse einer anderen Variablen vom Typ Kunde enthält.

Als Beispiel soll eine lange Liste `KundenListe()` des Typs `.Kunde` dienen, die nach dem Kunden mit dem größten Umsatz durchsucht werden soll. Wenn der Umsatz eines Kunden (also eines Listenelements), größer ist als der, auf den die Variable `groesster` im Moment zeigt, wird diese so verändert, daß sie auf den aktuellen Kunden (also das aktuelle Listenelement) zeigt, wie in der folgenden Zeile:

```
*groesster=KundenListe()
```

Wenn in einer Schleife die ganze Liste durchlaufen ist, kann die Variable `groesster` ausgedruckt werden, so als ob sie selber vom Datentyp `.Kunde` wäre, obwohl sie eigentlich nur ein Zeiger auf eine andere Variable diese Typs ist:

```
Print *groesster\name
```

Kapitel 4

Prozeduren

- Einleitung
- Befehle
- Funktionen
- Rekursion
- Globale Variablen
- Zusammenfassung

Einleitung

Prozeduren dienen dazu, Programmabschnitte (Routinen) in abgeschlossene Module zu „verpacken“.

Routinen, die zu Prozeduren verpackt wurden, können vom Hauptprogramm aus aufgerufen werden, Parameter können übergeben werden und es kann auch ein Wert an das aufrufende Programm zurückgegeben werden.

Da eine Prozedur ihre eigenen lokalen Variablen verwaltet, kann es keine Verwechslung zwischen den lokalen Variablen der Prozedur und den Variablen des Hauptprogramms geben. Ebensowenig kann eine Prozedur die Variablen des Hauptprogramm verändern. Dadurch werden Prozeduren „portabel“, d.h. sie können in anderen Programmen wiederverwendet werden.

Prozeduren, die einen Wert zurückliefern, werden Funktionen genannt, die anderen heißen Befehle.

Für Funktionen und Befehle gelten folgende Regeln:

- die Anzahl der Parameter ist auf 6 begrenzt
- `Goto`- und `Gosub`-Befehle dürfen nie zu Sprungmarken (Labels) außerhalb der eigenen Prozedur springen
- alle Variablen, die innerhalb der Prozedur verwendet werden, werden bei jedem Aufruf neu initialisiert

Befehle

Eine Prozedur, die keinen Wert an das aufrufende Programm zurückliefert heißt in Blitz2 Befehl (engl. Statement).

Hier ist ein Beispiel einer Befehls-Prozedur, die die Fakultät einer Zahl ausgibt:

```
Statement fact{n}
      a=1
      For k=2 To n
            a=a*k
      Next
      NPrint a
End Statement
For k=1 To 5
      fact{k}
```

```
Next
MouseWait
```

Sowohl bei der Definition der Prozedur als auch bei deren Aufruf werden die Parameter in geschweiften Klammern *{}* eingeschlossen. Die Klammern müssen auch dann angegeben werden, wenn keine Parameter übergeben werden sollen.

Dieses Programm berechnet die Fakultäten der Zahlen von 1 bis 5 in einer Schleife. Dabei wird im Hauptprogramm die Variable `k` als Schleifenindex verwendet. Gleichzeitig wird in der Prozedur ebenfalls eine Variable mit dem Namen `k` verwendet. Dies ist deshalb zulässig, weil die Variable `k` der Prozedur `fact` dort lokal vereinbart wurde und völlig unabhängig von der Variablen `k` im Hauptprogramm ist. Die Größe `k` des Hauptprogramms wird globale Variable genannt.

Es können bis zu sechs Variablen als Parameter an eine Prozedur übergeben werden. Wenn mehr als sechs Parameter benötigt werden, können diese als spezielle „shared“ globale Variable vereinbart werden (siehe „Globale Variablen“)

Variablen, die an eine Prozedur als Parameter übergeben werden, dürfen nur einfache Datentypen sein, `NewType`-Variablen können nicht übergeben werden. Wohl aber können Zeiger-Variablen übergeben werden.

Funktionen

Blitz2 erlaubt es auch, Prozeduren zu schreiben, die einen Wert an das Hauptprogramm zurückliefern. Solche Prozeduren werden Funktionen genannt. Das folgende Beispiel zeigt dieselbe Prozedur für Fakultäten, diesmal aber als Funktion realisiert:

```
Function fact{n}
      a=1
      For k=2 To n
            a=a*k
      Next
      Function Return a
End Function
For k=1 To 5
      NPrint fact{k}
Next
MouseWait
```

Durch die Anweisung `Function Return` wird das Ergebnis an das Hauptprogramm zurückgeliefert. Diese Version von `fact` ist wesentlich nützlicher als die erste, da das Ergebnis wiederum in beliebigen Ausdrücken verwendet werden kann. Z.B.:

```
a=fact{k}*fact{j}
```

Es können alle 6 einfachen Datentypen als Rückgabewert verwendet werden. Um klarzumachen, welchen Datentyp eine Prozedur zurückgeben soll, wird der Typ-Zusatz an den Befehl Function angefügt. Ist kein Typ angegeben, wird der Default-Typ angenommen (normalerweise *.q*). Im folgenden Beispiel wird ein *String* zurückgeliefert:

```
Function$ leerz{n}
      For k=2 To n
            a$=a$+" "
      Next
      Function Return a$
```

```
End Function
Print leerz{20}, "Hier drüben"
MouseWait
```

Rekursion

Der Speicherbereich, der von den lokalen Variablen einer Prozedur belegt wird, ist nicht nur allein für diese Prozedur reserviert, sondern auch für jeden Aufruf der Prozedur. Jedesmal, wenn die Prozedur aufgerufen wird, wird auch ein neuer Speicherbereich angelegt, der erst dann wieder freigegeben wird, wenn die Prozedur beendet ist.

Dadurch ist es möglich, daß eine Prozedur sich selbst aufrufen kann, ohne sich dabei die eigenen Daten zu zerstören. Dieses Vorgehen nennt man Rekursion. Das Programm zur Berechnung der Fakultät ist ein gutes Beispiel dafür, wie die Rekursion sinnvoll eingesetzt werden kann:

```
Function fact{n}
      If n>2 Then n=n*fact{n-1}
      Function Return n
End Function
For k=1 To 5
      NPrint fact{k}
Next
MouseWait
```

Das Prinzip dieses Algorithmus beruht darauf, daß die Fakultät einer Zahl gerade die Zahl selbst ist, multipliziert mit der Fakultät der um 1 erniedrigten Zahl.

Globale Variablen

Manchmal ist es notwendig, daß eine Prozedur auch auf die globalen Daten des Hauptprogramms zugreift. Dies wird mit der Anweisung „Shared“ erreicht, mit der es möglich ist, bestimmte Variablen innerhalb einer Prozedur als globale Variablen zu behandeln.

```
Statement beisp{}
      Shared k
      NPrint k
End Statement
For k=1 To 5
      beisp{}
Next
MouseWait
```

Durch die Anweisung `Shared` wird erreicht, daß Blitz2 in der Prozedur `beisp` keine eigene lokale Variable `k` anlegt, sondern stattdessen auf die globale Variable `k` des Hauptprogramms zugreift. Ohne die `Shared`-Anweisung wird `k` zu einer lokalen Variable und wird somit bei jedem Aufruf von `beisp` neu initialisiert, der Wert ist dann jedesmal 0.

Zusammenfassung

Blitz2 unterstützt zwei Arten von Prozeduren: Befehle und Funktionen. Beide können sowohl eigene lokale Variablen besitzen als auch mit Hilfe der `Shared`-Anweisung auf globale Daten zugreifen.

Es können bis zu sechs Parameter an eine Prozedur übergeben werden.

Eine Funktion kann einen der sechs einfachen Datentypen als Wert zurückliefern.

Kapitel 5

Beispiele

- Zahlen raten
- Ein eigenständiges Workbench-Programm
- Ein graphisches Beispiel
- Menüs & File-Requester
- String-Gadgets
- Prop-Gadgets
- Telefonbuch-Datenbank
- Verwaltung von Betriebssytem-Listen
- Primzahlgenerator

In diesem Kapitel werden die in den vorhergehenden Kapiteln beschriebenen Konzepte anhand zahlreicher Beispiele erläutert.

Unerfahrene Programmierer sollten auf jeden Fall zunächst die Beispiele in Kapitel 2 ausprobieren, bevor Sie sich an diese heranwagen.

Zahlen raten

Im folgenden kleinen Programm denkt sich der Rechner eine Zahl aus und der Benutzer muß versuchen, Sie in zehn versuchen zu raten.

```
NPrint "Ich habe mir eine Zahl zwischen 0 und 100 ausgedacht"
NPrint "Ich gebe dir zehn Versuche, sie zu raten:"

a=Rnd(100)
n=1

Repeat
      Print n, ". Versuch ?"
      b=Edit(10)
      If b=a Then NPrint "Glück gehabt":Goto finish
      If b<a Then NPrint "Zu klein"
      If b>a Then NPrint "Zu gross"
      n+1
Until n=11

NPrint "10 Versuche, das war's!"

finish:

NPrint "Mausknopf drücken um das Programm zu beenden."
MouseWait
```

Es wird zunächst recht schwierig sein, die Zahl zu raten, weil der Zufallszahlengenerator standardmäßig keine ganze Zahl erzeugt, sondern auch Nachkommastellen.

Ändern Sie daher die Zeile `a=Rnd(100)` entweder in `a.w=Rnd(100)` oder `a=Int(Rnd(100))`.

Der Zusatz `.w` steht für *Word* und hat zur Folge, daß die Variable `a` jetzt ganzzahlig (im Bereich -32768 bis +32767) ist. Bei der zweiten angegebenen Möglichkeit ist `a` zwar immer noch vom Typ *Quick*, durch den Aufruf von `Int()` wird der Nachkommateil der Zahl jedoch abgeschnitten.

Immer wenn in Blitz2 eine Variable ohne *.typ*-Suffix angegeben wird, wird automatisch der *Quick*-Typ angenommen. Der Wertebereich liegt zwischen -32768 und +32767 bei einer Genauigkeit von 1/65536. Siehe auch den Abschnitt über Datentypen.

Wenn alle Variablen des Programms standardmäßig vom Typ *Word* sein sollen, müssen Sie folgende Zeile am Anfang des Programms einfügen:

```
DEFTYPE .w  ; alle Variablen ohne Suffix sind Word
```

Ist eine Variable einmal vereinbart worden, ist ihr Typ damit festgelegt und eine Angabe des *.typ*-Zusatzes ist bei weiteren Zugriffen nicht mehr nötig.

Im Gegensatz zu anderen BASIC-Versionen fügt das `Print`-Kommando nicht automatisch einen Zeilenvorschub an das Ende des ausgegebenen Textes an. Wenn dieses gewünscht wird, ist der `NPrint`-Befehl zu verwenden.

Die Funktion `Edit()` ersetzt das ältere Kommando `Input`. Ebenso wird das Semikolon statt des Befehls `REM` für Kommentare verwendet und es besitzt nicht mehr die alte Bedeutung in Zusammenhang mit dem `Print`-Befehl.

Ein eigenständiges (standalone) WorkBench-Programm

Aus dem Rate-Programm kann ein eigenständiges Programm mit zugehörigem Symbol(Icon) gemacht werden, daß von der Workbench aus gestartet werden kann.

Die folgenden Zeilen sind am Anfang des Programmcodes einzufügen. Die Texte hinter dem Semikolon sind, wie bereits erwähnt, Kommentare.

```
;
; Zahlen-Rate Programm
;
WBStartup   ; noetig um Prog. von der Workbench aus zu starten
FindScreen 0 ; vorderste Intuition Screen holen
;
Window 0,0,0,320,210,$1000,"Hello World",1,2
```

Wenn Sie das Programm jetzt übersetzen und laufen lassen, wird die Ein- und Ausgabe über das Window erledigt, anstatt mit der Kommandozeile.

Es sollte noch folgende Änderung der Zeile `b=Edit(10)` vorgenommen werden:

```
b=Val(Edit$(10))
```

Dadurch wird die standardmäßige Ausgabe des 0-Zeichens bei der Window-Version von `Edit()` unterdrückt.

Schalten Sie jetzt die Option „Create Executable Icon“ im Editor-Menü an und wählen Sie „Create Executable“ oder drücken Sie das Kürzel *Amiga-E*.

Geben Sie dann einen Namen für das Programm ein. Sie haben soeben ihr erstes eigenständiges Workbench-Programm erzeugt. Dies können Sie jetzt starten, indem Sie auf das zugehörige Icon klicken.

Ein graphisches Beispiel

Das folgende Programm öffnet zunächst eine eigene Screen und zeichnet dann eine Rosette, also eine Figur, bei der alle Punkte eines Kreises durch Linien verbunden werden.

```
;
; Rosetten-Beispiel
;

n=20

NEWTYPE .pt
      x.w:y
End NEWTYPE

Dim p.pt(n)

For i=0 To n-1
      p(i)\x=320+Sin(2*i*Pi/n)*319
      p(i)\y=256+Cos(2*i*Pi/n)*255
Next

Screen 0,25 ;einfarbige interlace screen anfordern
ScreensBitMap 0,0

For i1=0 To n-2
      For i2=i1+1 To n-1
            Line p(i1)\x,p(i1)\y,p(i2)\x,p(i2)\y,1
      Next
Next

MouseWait
```

Der mit `NewType` definierte Datentyp .pt besitzt zwei Elemente, `x` und `y`. Dadurch brauchen nicht zwei Felder `x.w(n)` und `y.w(n)` angelegt zu werden, sondern nur ein Feld des Typs `.pt`, das dieselbe Information enthält.

Die einzelnen Elemente des `NewType` werden mit dem Backslash-Zeichen „\“ erreicht.

In der ersten `For...Next`-Schleife werden zunächst die Punkte des Kreises berechnet.

Mit dem Befehl `ScreenBitMap` kann anschließend direkt auf die Screen mit den Befehlen `Plot`, `Line`, `Box` und `Circle` zugegriffen werden. Programme, die Windows benutzen, sollten diese Methode nicht anwenden, sondern zum Zeichnen die Befehle `WPlot`, `Wline`, `WBox` und `WCircle` verwenden.

Menüs und File-Requester

Das folgende Beispiel demonstriert den Gebrauch von Menüs und File-Requestern. Es erzeugt eine Screen und ein Window, bringt eine Menüleiste an und je nachdem, was der Benutzer auswählt, erscheint ein File-Requester oder das Programm wird beendet.

```
;
; Ein einfaches Beispiel für einen File Requester
;
Screen 0,11,"Ein Menü auswählen"    ; intuition screen öffnen

MenuTitle 0,0,"Projekt"             ; Menüliste anlegen
MenuItem 0,0,0,0,"Laden       ","L"
MenuItem 0,0,0,1,"Sichern     ","S"
MenuItem 0,0,0,2,"Quit        ","Q"

MaxLen path$=192  ;MUSS angegeben werden, bevor der Requester benutzt wird
MaxLen name$=192

; Ein BACKDROP (d.h. unsichtbares) Window erzeugen
Window 0,0,0,320,200,$1900,"",1,2
WLocate 0,20      ; Cursor in die linke obere Ecke
SetMenu 0         ;Menü mit dem Fenster verbinden

Repeat
      Select WaitEvent
            Case 256                ; ein Menü-Event!
                  Select ItemHit

                  Case 0            ; Eintrag 0: Laden
                        p$=FileRequest$("Datei laden",path$,name$)
                        NPrint "Jetzt wird ",p$, "geladen"

                  Case 1            ; Eintrag 1: Sichern
                        p$=FileRequest$("Datei sichern", path$, name$)
                        NPrint "Jetzt wird ",p$, "gesichert"

                  Case 2            ;Eintrag 2:Quit
                        End

            End Select

      End Select
Forever
```

Mit dem Befehl `MaxLen` wird ein Speicherbereich für eine Stringvariable angefordert. Dies ist notwendig, damit die beiden Stringvariablen, für den File-Requester genügend Platz für die Eingabe haben.

Die Menüs, die mit MenuTitle und MenuItem erzeugt wurden, werden mit dem Befehl `SetMenu` mit dem Window verbunden.

Die Struktur `Select...Case...EndSelect` eignet sich am besten, um auf Ereignisse, die von außen (in diesem Fall vom Benutzer) kommen zu reagieren. Wenn der Benutzer ein Menü auswählt, ein Window schließt oder auf einen Knopf klickt, findet ein Ereignis statt und eine entsprechende Nachricht wird an das Programm gesendet. Ein Programm sollte mit dem Befehl `WaitEvent` auf ein Ereignis warten. Dadurch „schläft“ das Programm solange, bis der Benutzer etwas eingibt. In einer Multi-Tasking-Umgebung können

dann in der Zwischenzeit andere Programme weiterarbeiten. Ein Programm, das auf ein Ereignis wartet, belastet das System nicht.

Wurde ein Event empfangen, liefert `WaitEvent` eine Kennung für die Art des Events zurück. Ein Menü-Ereignis liefert z.B. den Wert 256 ($100 hex) zurück, ein Close Winwow Ereignis liefert 512 ($200 hex). Auf Seite 25-5 <<< ? >>> des Referenzhandbuchs befindet sich eine vollständige Liste der Events.

String Gadgets

String Gadgets sind Eingabefelder, in die der Benutzer Texteingaben mit der Tastatur vornehmen kann. Im folgenden Beispiel werden 3 Eingabefelder für dezimale, hexadezimale und binäre Ein- und Ausgabe erzeugt.

Wenn der Benutzer eine Zahl in eines der Gadgets einträgt, erhält das Programm eine Gadgetup-Nachricht. Die Funktion `GadgetHit` liefert dann die Nummer des Gadgets, in dem die Eingabe erfolgte. Anschließend konvertiert das Programm die Zahl in die anderen beiden Zahlensysteme und gibt das Ergebnis in allen Gadgets aus.

Mit dem Befehl `ActivateString` wird erreicht, daß der Benutzer nicht erst mit der Maus auf eines der Gadgets klicken muß, bevor etwas eingegeben werden kann.

```
;
; Dezimal, Hex, Binär Konverter
;

FindScreen 0

StringGadget 0,64,12,0,0,18,144
StringGadget 0,64,26,0,1,18,144
StringGadget 0,64,40,0,2,18,144

Window 0,100,50,220,56,$1008,"BASIS KONVERTER",1,2,0

WLocate 2,04:Print "DEZIMAL"
WLocate 2,18:Print "   HEX$"
WLocate 2,32:Print "BINÄR%"

DEFTYPE.l value

Repeat
      ev.l=WaitEvent
      If ev=$40                          ;Gadget up Event
            Select GadgetHit
                  Case 0
                        value=Val(StringText$(0,0))
                  Case 1
                        r$=UCase$(StringText$(0,1))
                        value=0:i=Len(r$):b=1
                        While i>0
                              a=Asc(Mid$(r$,i,1))
                              If a>65 Then a-55 Else a-48
                              value+a*b
                              i-1:b*16
                        Wend
                  Case 2
```

```
                        r$=StringText$(0,2)
                        value=0:i=Len(r$):b=1
                        While i>0
                              a=Asc(Mid$(r$,i,1))-48
                              value+a*b
                              i-1:b*2
                        Wend
                End Select
                ActivateString 0,GadgetHit
                SetString 0,0,Str$(value)
                SetString 0,1,Right$(Hex$(value),4)
                SetString 0,2,Right$(Bin$(value),16)
                Redraw 0,0:Redraw 0,1:Redraw 0,2
        EndIf
Until ev=$200
```

Prop Gadgets

Das folgende Beispielprogramm erzeugt einen einfachen Requester für eine RGB-Palette, mit dem die Bildschirmfarben eingestellt werden können. Prop Gadgets sind Schieberegler, die in diesem Fall für die Einstellung der Rot-, Grün- und Blaukomponenete des aktuellen Farbregisters dienen.

Die 32 Farbregister werden durch 32 Text Gadgets repräsentiert. Die Farbe der einzelnen Gadgets wird mit dem Befehl `GadgetPen` festgelegt. Dies geschieht bevor die Gadgets zu der Objektliste hinzugefügt werden. Der Befehl `GadgetJam` führt dazu, daß die beiden Leerzeichen als farbige Blöcke dargestellt werden.

```
;
; Einfache Farbpalette
;

FindScreen 0

For p=0 To 2
      PropGadget 0,p*22+8,14,128,p,16,54
Next

For c=0 To 31
      GadgetJam 1:GadgetPens 0,c
      x=c AND 7:y=Int(c/8)
      TextGadget 0,x*28+72,14+y*14,32,3+c,"  " ;<-2 Leerzeichen
Next

Window 0,100,50,300,72,$100A,"FARB-PALETTE",1,2,0

cc=0:Toggle 0,3+cc,On:Redraw 0,3+cc

Repeat
      SetVProp 0,0,1-Red(cc)/15,1/16
      SetVProp 0,1,1-Green(cc)/15,1/16
      SetVProp 0,2,1-Blue(cc)/15,1/16
      Redraw 0,0:Redraw 0,1:Redraw 0,2
      ev.l=WaitEvent
      If ev=$40 AND GadgetHit>2
            Toggle 0,3+cc,On:Redraw 0,3+cc
            cc=GadgetHit-3
            Toggle 0,3+cc,On:Redraw 0,3+cc
```

```
        EndIf
        If (ev=$20 OR ev=$40) AND GadgetHit<3
              r.b=VPropPot(0,0)*16
              g.b=VPropPot(0,1)*16
              b.b=VPropPot(0,2)*16
              RGB cc,15-r,15-g,15-b
        EndIf
    Until ev=$200
```

Datenbank-Anwendung

Es folgt ein Beispiel für ein einfaches Datenbank-Programm, eine Adressverwaltung. Es verwaltet eine Liste mit Namen, Telefonnummern und Adressen.

Die Definition der Bedienoberfläche kann entweder so eingetippt werden, wie sie weiter unten gedruckt ist, oder sie kann mit IntuiTools erstellt werden.

Wenn eine Datei mit dem Namen *phonebook.data* existiert, wird sie eingelesen. Es wird eine Liste daraus aufgebaut, deren Einträge aus jeweils 4 Strings bestehen, die im NewType definiert sind.

Durch die Verwendung einer Liste statt eines einfachen Feldes, wird klar, daß die Einträge in einer gewissen Beziehung zueinander stehen. Blitz2 verwaltet einen internen Zeiger auf das aktuelle Listenelement, der mit einer Reihe von Listenbefehlen gesetzt und abgefragt werden kann.

```
;
; Telefonbuch-Programm
;

FindScreen 0

; Die folgende Beschreibung wird von ram:t gelesen und wurde mit
; IntuiTools erstellt

Borders On:BorderPens 1,2:Borders 4,2
StringGadget 0,72,12,0,1,40,239
StringGadget 0,72,27,0,2,40,239
StringGadget 0,72,43,0,3,40,239
StringGadget 0,72,59,0,4,40,239
GadgetJam 0:GadgetPens 1,0
TextGadget 0,8,75,0,10,"NEUER EINTRAG"
TextGadget 0,97,75,0,11,"|<"
TextGadget 0,129,75,0,12,"<<"
TextGadget 0,161,75,0,13,">>"
TextGadget 0,193,75,0,14,">|"
TextGadget 0,226,75,0,15,"DIAL"
TextGadget 0,270,75,0,16,"LABEL"

SizeLimits 32,32,-1,-1
Window 0,0,24,331,91,$100E,"MEIN TELEFONBUCH",1,2,0
WLocate 2,19:WJam 0:WColour 1,0
Print "Adresse"
WLocate 19,50
Print "Telefon"
WLocate 27,3
```

```
Print "Name"

; nun beginnt die Eingabe...

#num=4      ;4 strings für jede Person

NEWTYPE .person
 info$[#num]
End NEWTYPE

Dim List people.person(200)

USEPATH people()

; Daten aus bestehendem sequentiellen File einlesen

If ReadFile (0,"phonebook.data")
      FileInput 0
      While NOT Eof(0)
            If AddItem(people())
                  For i=0 To #num-1:\info[i]=Edit$(128):Next
            EndIf
      Wend
EndIf

ResetList people()

; Wenn Liste leer, ein Listenelement einfügen

If NOT NextItem(people()) Then AddItem people()


refresh:
      ref=0
      For i=0 To #num-1
            SetString 0,i+1,\info[i]:Redraw 0,i+1
      Next
      ActivateString 0,1:VWait 5
      Repeat
            ev.l=WaitEvent
            ;
            If ev=$200
                  ; Fenster schliessen
                  Gosub update
                  If WriteFile (0,"phonebook.data");Daten in File schreiben
                        FileOutput 0
                        ResetList people()
                        While NextItem(people())
                              For i=0 To #num-1:NPrint \info[i]:Next
                        Wend
                        CloseFile 0
                  EndIf
            EndIf
            ;
            If ev=64
            If GadgetHit=#num Then ActivateString 0,1
            If GadgetHit<#num Then ActivateString 0,GadgetHit+1
                  Select GadgetHit
                        Case 10
```

```
                        Gosub update:If AddItem(people()) Then ref=1
                      Case 11
                        Gosub update:If FirstItem(people()) Then ref=1
                      Case 12
                        Gosub update:If PrevItem(people()) Then ref=1
                      Case 13
                        Gosub update:If NextItem(people()) Then ref=1
                      Case 14
                        Gosub update:If LastItem(people()) Then ref=1
                End Select
          EndIf
      Until ref=1
      Goto refresh
update:
      For i=0 To #num-1:\info[i]=StringText$(0,i+1):Next:Return
```

Verwaltung von Betriebssystem-Listen

Das folgende Programm erläutert den Zugriff auf Strukturen des Betriebssystems. Dieses Programm benötigt die Datei *AmigaLibs.res* in der Blitz2-Umgebung. Hierfür wählen Sie die „Compiler Options“ aus dem Menü, klicken in das Feld „Residents“ und geben den Namen *AmigaLibs.res* ein. Möglicherweise wird ein Pfadname benötigt. Die Datei AmigaLibs befindet sich im Verzeichnis „Residents“ der Blitz2 Programmdiskette.

Wenn Sie jetzt „View Types“ im Compiler-Menü auswählen, erscheint eine Liste aller Strukturen, die vom Amiga-Betriebssystem verwendet werden.

In der ersten Programmzeile wird eine Variable `exec` als Zeiger vom Typ `ExecBase` definiert. Da der Amiga die Adresse der Listen in der Speicheradresse 4 hält, kann mit dem Befehl `Peek.l(4)` der 4 Byte lange Wert vom Speicher in unsere Zeigervariable gelesen werden.

Da die Variable `exec` als Zeiger auf eine `ExecBase`-Struktur definiert wurde, kann jetzt mit dem Backslash auf jedes Element der Struktur mit dessen Namen zugegriffen werden. Die Namen der Strukturelemente können mit Hilfe des ViewType-Menüs angesehen werden, wenn Sie den Namen ExecType (Großschreibung beachten) eingeben.
Anschließend wird eine weitere Pointervariable angelegt (diesmal vom Typ *.List*), die auf ein Listenelement der Exec-Struktur zeigt.

Alle Exec-Listen bestehen aus Knoten (nodes). Der erste Knoten repräsentiert immer den Listenkopf, danach folgen die Daten.

In der dritten Zeile wird die Variable `mynode` als Zeiger vom Typ *.Node* definiert und zeigt auf den ersten Knoten der Liste.

In der Schleife wird die Liste dann solange abgearbeitet, bis der Zeiger auf den Nachfolger (successor) 0 ist. Damit ist wieder der Anfang der Liste erreicht.

Der Befehl `Peek$` liest ASCII-Zeichen aus dem Speicher, bis eine Null erreicht wird. Dies ist sehr nützlich um Strings, die in C-Notation abgelegt sind (C-Strings werden mit einem Null-Byte beendet), wie `*In_Name`, einzulesen.

Als letztes wird dem Pointer `mynode` die Adresse des eigenen Nachfolgers zugewiesen.

```
      ;
```

```
; Exec-Listenverwaltung
;

*exec.ExecBase=Peek.l(4)

*mylist.List=*exec\LibList

*mynode.Node=*mylist\lh_Head

While *mynode\ln_Succ
 a$=Peek$(*mynode\ln_Name)
 NPrint a$
 *mynode=*mynode\ln_Succ
Wend

MouseWait
```

Primzahlgenerator

Im letzten Beispiel wird eine Liste aller Primzahlen von 2 bis zu einer Höchstzahl, die vom Benutzer eingegeben wird, berechnet. Die Primzahlen werden in Form einer Blitz2-Liste verwaltet.

Das Programm beginnt mit dem Einlesen der Höchstgrenze mit Hilfe der Standard Eingabe- und Ausgabekanäle und des Befehls `Edit()`, der numerischen Variante von `Edit$()`.

Die Arbeitsschleife `While..Wend` wird solange durchlaufen, bis der Höchstwert erreicht ist. Der Algorithmus besteht lediglich darin, die nächste ganze Zahl zu nehmen und mit der Liste der zuvor erzeugten Primzahlen zu vergleichen, bis eine ganzzahlige Division möglich ist oder bis die Prüfgrenze erreicht wurde (dies ist die Wurzel der in Frage kommenden Zahl).

Wurde kein ganzzahliger Teiler gefunden, wird die neue Primzahl ausgedruckt und an das Ende der Liste angefügt.

```
Print "Höchstwert eingeben "  ; Grenzwert des Programms
v=Edit(80)                    ; numerische Eingabe
If v=0 Then End               ; Abbruch bei 0
tab.w=0:tot.w=0               ; Zähler rücksetzen
Dim List primes(v)            ; Primzahl-Liste anlegen
p=2                           ; 2 ist der erste Eintrag
AddItem primes()
primes()=p

While p<v                     ; Schleife bis Ende erreicht ist
       p+1                    ; p erhöhen
       flag=1                 ; flag setzen
       d=0
       q=Sqr(p)               ; Suchgrenze festlegen
       ResetList primes()     ; Schleife über die Liste
       While NextItem(primes()) AND d<q AND flag
                 d=primes()
                 flag=p MOD d
       Wend
       If flag<>0             ; gefunden: ausgeben und in Liste eintragen
                 Print p,Chr$(9)          ;chr$(9) ist der TAB
                 tab+1:tot+1
```

```
                If tab=10 Then NPrint "":tab=0
                AddLast primes()
                primes()=p
        EndIf
Wend

NPrint Chr$(10)+"Habe ",tot," Primzahlen zwischen 2 & ",v, "gefunden"
NPrint "Beenden mit linkem Mausknopf"

MouseWait
```

Kapitel 6

Fehlermeldungen & der Debugger

- Fehlermeldungen des Compilers
- Laufzeitfehler
- Der Blitz2-Debugger
- Anzeige-Optionen
- Verfolgung des Programmablaufs
- Wiederaufnahme des normalen Ablaufs
- Der Befehlspuffer
- Der Direktmodus
- Fehlermeldungen des Debuggers

Fehlermeldungen des Compilers

Blitz2 erzeugt zwei unterschiedliche Kategorien von Fehlermeldungen: Übersetzungsfehler und Laufzeitfehler. Übersetzungsfehler treten während des Compilierens (Übersetzen) des Quellcodes auf, Laufzeitfehler treten während der Ausführung (Run time) des Programms auf.

Tritt während des Compilierens ein Fehler auf, so erscheint eine Nachricht im Editor. Wird *OK* angeklickt, so wird zu der Zeile im Programm zurückgekehrt, die den Fehler hervorgerufen hat.

In Anhang 2 des Blitz2 Referenz-Handbuchs befindet sich eine vollständige Beschreibung aller Compilermeldungen. Im folgenden wird auf einige der häufigsten Fehlerquellen hingewiesen:

· Beim Aufruf einer internen Blitz2-Funktion (also eines Befehls, der einen Wert zurückliefert) müssen die Parameter immer in Klammern eingeschlossen werden:

```
If ReadFile (0,"ram:test")
```

· Im Gegensatz dazu dürfen beim Aufruf von internen Blitz2-Befehlen, die keine Funktionen sind, die Parameter nicht in Klammern erscheinen:

```
BitMap 0,320,256,3
```

· Die Angabe eines Typ-Zusatzes beim Zugriff auf Elemente eines `NewType` führt zu der Meldung „Garbage at end of line“ („Schrott am Ende der Zeile“):

```
person\name$="Harry"    ; zur Fehlerbehebung  das $-Zeichen weglassen
```

· Der Daten-Typ einer numerischen Variable kann sich nicht ändern. Die Angabe eines anderen Typ-Zusatzes als des ursprünglichen führt zu der Meldung „MisMatched Type“ („Nicht-gleichwertige Datentypen“). Eine Ausnahme bilden String-Variablen.

Es gibt natürlich hunderte von möglichen Fehlern, die zu einem Abbruch des Compilierens führen, aber die meisten sind auf einfache Syntax-Fehler zurückzuführen und lassen sich schnell durch einen Blick in das Referenz-Handbuch klären. Oft ist auch ein Vergleich mit den Beispielprogrammen hilfreich.

Außerdem gibt es noch die Help-Taste, die eine schnelle Überprüfung der Befehls-Syntax ermöglicht.

Die Anweisung `CERR`

Bei dem Gebrauch von Makros und bedingter Compilierung („conditional compiling“) kann es nützlich sein, eigene Compilermeldungen zu erzeugen.

Hierzu wird die `CERR`-Anweisung benutzt. Das folgende Beispiel führt dazu, daß der Compiler anhält und die Meldung „Muß 3 Parameter haben“ ausgibt:

```
CERR "Muß 3 Parameter haben"
```

Im Kapitel 9 ist das bedingte Compilieren und die `CERR`-Anweisung eingehender beschrieben.

Laufzeitfehler

Unter Laufzeitfehlern („run time error“) versteht man solche Fehler, die während der Ausführung des Programms auftreten.

Während der Entwicklungsphase eines Programms sollte der „Runtime Error Debugger“ immer aktiviert sein. Dies geschieht über das „Compiler Options“ Menü. Ist der Debugger nicht aktiviert und es tritt ein Laufzeitfehler auf, stürzt das System ab.

Falls aus Geschwindigkeitsgründen die Überprüfung der Laufzeitfehler ausgeschaltet sein muß, sollte eine `SetErr`-Anweisung eingefügt werden, um einen Systemabsturz zu vermeiden. Das System kehrt dann zu der Programmzeile hinter der `SetErr`-Anweisung zurück.

Es ist empfehlenswert, die folgende Zeile am Anfang des Programms einzufügen:

```
SetErr:End:End SetErr
```

Eine der häufigsten Fehlerursachen ist der Zugriff auf Dateien, die entweder nicht vorhanden oder vom falschen Typ sind. Daher sollten Programme, die auf Dateien zugreifen, grundsätzlich immer eine der genannten Methoden der Fehlererkennung verwenden.

Ebenso sollten alle Programme, die Betriebssystem-Aufrufe verwenden, immer die Fehlerüberprüfung benutzen, da Bildschirm- und Window-Zugriffe manchmal an Speichermangel scheitern.

Ein „Error Handler“ (Fehlerbehebungs-Routine) kann auch auf einen bestimmten Programmabschnitt beschränkt werden, dazu wird die Sequenz `SetErr...errohandler...EndSetErr` am Anfang und `ClrErr` am Ende des Abschnitts eingefügt.

Im folgenden Beispiel blinkt der Bildschirm auf, wenn der Aufruf von `LoadShapes` scheitert:

```
SetErr
      DisplayBeep_ 0
      End
EnsSetErr
LoadShapes 0, "dateiname"
ClrErr
```

Der Blitz2-Debugger

Tritt ein Laufzeitfehler in einem Programm auf, das vom Editor aus gestartet wurde, so wird der Debugger aktiviert, vorausgesetzt er wurde im „Compiler Options“ Menü eingeschaltet.

Ist in dem Programm selbst schon ein Error Handler aktiv, so wird der Debugger nicht aktiviert.

Der Debugger kann auch durch die *Ctrl/Alt-C*-Tastenkombination aktiviert werden. Außerdem dient der `STOP`-Befehl dazu, das Programm zu unterbrechen und den Debugger aufzurufen.

Der Debugger ist ein unentbehrliches Werkzeug bei der Fehlersuche. Seine Fähigkeit, von einer Stelle aus rückwärts durch den Programmcode zu gehen, der vorher ausgeführt wurde, bietet eine ausgezeichnete Möglichkeit, Fehler zu lokalisieren.

Alle Befehle des Debuggers werden aufgerufen, indem die *Ctrl*- und die linke *Alt*-Taste gleichzeitig mit der Taste für das Kommando gedrückt werden. Es gibt zwei verschiedene Arten von Kommandos: solche die immer während des Programmlaufs verfügbar sind und solche, die nur dann verfügbar sind, wenn das Programm gestoppt wurde.

Wenn ein Laufzeitfehler auftritt, wird automatisch das „Keylock“ eingeschaltet. Keylock bedeutet, daß die *Ctrl/Alt*-Kombination nicht mehr betätigt zu werden braucht um die Debugger-Kommandos aufzurufen (näheres s.u. unter K der nachfolgenden Liste).

Folgende Kommandos sind grundsätzlich immer verfügbar:

C - Programm anhalten

H - *Hide*: Debugger-Fenster zeigen/verstecken

V - *View*: Graphiken hinter dem Debugger-Fenster sichtbar machen oder verdecken

M - *Mode*: Umschalten zwischen hochauflösendem (hires) und niedrigauflösendem (lores) Graphikmodus wenn V aktiviert wurde

Die folgenden Kommandos sind nur dann verfügbar, wenn das Programm angehalten wurde:

Q - *Quit*: Programm beenden, auch über die *ESC*-Taste (hat dieselbe Auswirkung wie der Befehl `END`).

R - *Run*: Programm normal ablaufen lassen.

T - *Trace*: Ablaufverfolgung starten.

S - *Single Step*: Programm im Einzelschritt-Modus laufen lassen.

I - *Ignore*: Den aktuellen Befehl überspringen.

L - Umschalten der Ebene in der Ablaufverfolgung für das **S**- und **T**-Kommando. Hiermit kann über Gosubs, Prozeduren und For..Next-Schleifen hinweggegangen werden.

B - *Back*: Zurückpositioneren im Befehlspuffer zum zuvor ausgeführten Befehl.

F - *Forward*: Vorwärtspositionieren im Befehlspuffer zum nächsten ausgeführten Befehl.

E - *Evaluate*: Den Wert eines Ausdrucks kontrollieren. Der Wert wird im Debugger-Fenster angezeigt.

X - *Execute*: Einen Befehl ausführen.

K - *Keylock*: Umschalten des Keylock-Modus. Ist der Keylock-Modus eingeschaltet, wird die *Ctrl/Alt*-Kombination nicht benötigt um Debugger-Kommandos einzugeben. Es genügt dann die entsprechende Kommando-Taste.

Anzeige-Optionen

Die Anzeige des Debuggers erscheint immer als vorderstes Fenster, über allen anderen „Screens“. Im Blitz-Modus erschient sie über allen „Slices“. Die Kommandos **H**, **V** und **M** beeinflussen die Anzeige-Optionen.

Das Kommando **H** (*Hide*) schaltet die Anzeige ein oder aus, **V** (*View*) schaltet die Anzeige auf „durchsichtig“, sodaß der Hintergrund ebenfalls gesehen werden kann.

Das Kommando **M** (*Mode*) ist nur von Bedeutung, wenn **V** aktiviert wurde. Es schaltet dann den Hintergrund zwischen hochauflösender und niedrigauflösender Graphikdarstellung um. Da im Transparent-Modus nur zwei Bit-Ebenen dargestellt werden können, ist die Darstellung nicht immer perfekt, reicht aber aus um einen Überblick zu erhalten.

Verfolgung des Programmablaufs

Der Debugger ermöglicht es, die Ausführung der einzelnen Befehle eines Programmes zu verfolgen. Dabei wird der zur Zeit ausgeführte Befehl im Debugger-Fenster angezeigt.

Das Kommando **S** (*Step*) dient dazu, im Einzelschritt-Modus durch das Programm zu gehen. Jedesmal wenn **S** gedrückt wird, führt der Debugger den Befehl aus, auf den der Pfeil gerade zeigt und hält wieder an.

T (*Trace*) läßt den Debugger die Befehle kontinuierlich ausführen und anzeigen. Mit der Eingabe von **C** kann das Programm angehalten werden.

Mit **L** (*Level*) wird die Ebene der Ablaufverfolgung umgeschaltet. Ist **L** aktiv, werden die Schritte einer `For..Next`-Schleifen nicht einzeln angezeigt, sondern die Schleife wird bis zu ihrem Ende ausgeführt. Außerdem werden Prozedur-Aufrufe wie ein normaler Befehl behandelt, es wird also nicht in Prozeduren „hineingesprungen“. Dies ist hilfreich, wenn z.B. die Arbeitsschleife des Hauptprogramms verfolgt werden soll, ohne daß die Ausführung der einzelnen Unterprogramme angezeigt wird.

Wiederaufnahme des normalen Ablaufs

Um das Programm normal weiterlaufen zu lassen, dient das Kommando **R** (*Run*).

Wenn der Debugger mit dem `STOP`-Befehl aktiviert wurde, zeigt der Pfeil auf die Zeile mit dem `STOP`. Dieser Befehl muß zunächst mit dem Kommando **I** (*Ignore*) übersrpungen werden. Das gleiche gilt für alle Befehle, die einen Laufzeitfehler verursacht und dadurch den Debugger aktiviert haben.

Um den Debugger zu verlassen und wieder in den Editor zurückzukehren, wird entweder das Kommando **Q** (*Quit*) oder die *ESC*-Taste benutzt.

Der Befehlspuffer

Der Debugger „merkt“ sich alle Befehle, die der Rechner ausgeführt hat bevor das Programm angehalten wurde, in einem speziellen Puffer. Mit den Kommandos **B** und **F** kann sich der Benutzer in diesem Puffer „bewegen“.

Mit dem Kommando **B** (*Backup*) können rückwärts die zuvor vom Rechner ausgeführten Befehle angesehen werden und zwar von der Stelle aus an der das Programman angehalten wurde. Die aktuelle Position in

diesem Puffer wird mit einem „Hohlpfeil“ angezeigt.

Das Kommando **F** (*Forward*) läßt den Benutzer in dem Befehlspuffer vorwärts gehen. Wenn versucht wird, über die Stelle, an der das Programm angehalten wurde, hinauszugehen, erscheint die Meldung „At End of Buffer“ („Ende des Puffers erreicht“).

Diese Möglichkeit, die vor dem Auftreten eines Fehlers ausgeführten Befehle nachträglich verfolgen zu können, ist von unschätzbarem Wert. Wenn ein Programm innerhalb eines Unterprogramms oder einer Prozedur gestoppt wurde, kann somit ermittelt werden, von welcher Stelle im Hauptprogramm aus die Routine gerufen wurde.

Der Direktmodus

Der Programmierer hat zwei Möglichkeiten, den internen Zustand des Programms zu kontrollieren und zu beeinflussen, wenn der Debugger aktiv ist.

Um herauszufinden, welchen Wert eine Variable gerade besitzt, wird das Kommando **E** (*Evaluate*) verwendet. Nach der Eingabe von **E** wird der Benutzer aufgefordert, den Variablennamen einzugeben. Anschließend wird der Wert der Variablen angezeigt.

Mit dem Kommando **X** (*Execute*) können Blitz2-Befehle ausgeführt werden. Nach der Eingabe von **X** erscheint ein Prompt und es kann ein beliebiger Befehl eingegeben werden, wie z.B. `CLS` oder `n=20`.

Fehlermeldungen des Debuggers

Bei der Ausführung der Kommandos **E** und **X** im Direktmodus können folgende Fehler auftreten:

„Can't create in Direct Mode“ („Kann im Direktmodus nicht erzeugt werden“)
: Diese Meldung erschient, wenn versucht wird, auf eine Variable zuzugreifen, die nicht existiert (erzeugt wurde).

„Library Not Available in Direct Mode“ („Bibliothek im Direktmodus nicht verfügbar“)
: Wenn ein Blitz2-Befehl ausgeführt werden soll, der aus einer Bibliothek stammt, die nicht mit dem Programm geladen wurde, tritt dieser Fehler auf. Wenn ein Programm beispielsweise keine Strings verwendet, dann ist die Bibliothek, die die String-Befehle enthält, nicht Teil des Objektcodes. Infolgedessen können auch mit dem **X**-Kommando keine String-Befehle ausgeführt werden.

„Not enough Room in Direct Mode Buffer“ („Kein Platz mehr im Puffer für Direktmodus“)
: Dieser Fehler sollte eigentlich nie auftreten. Tritt er dennoch auf, so muß die Größe des Objekt-Puffers im Menü „Compiler Options“ erhöht werden.

„AT END OF BUFFER“ („Ende des Puffers erreicht“)
: Diese Meldung erschient, wenn versucht wird, mit dem **F**-Kommando über das Ende des Befehlspuffers hinaus zu gehen (s. Befehlspuffer).

Kapitel 7

Blitz2 Objekte

- Übersicht
- Gemeinsamkeiten der Objekte
- Maximalwerte
- Benutzung eines Objekts
- Ein/Ausgabe-Objekte
- Objektstrukturen (für Fortgeschrittene)
- Überblick über die primären Blitz2-Objekte
    - Screens
    - Windows
    - Gadgets & Menülisten
    - Paletten
    - Bitmaps
    - Shapes
    - Sprites
    - Slices
    - Dateien
- Zusammenfassung

Überblick über die Blitz2-Objekte

In diesem Kapitel werden die Grundzüge von Blitz2-Objekten behandelt. Objekte in Blitz2 sind Strukturen, die dazu dienen, komplexe Systemeinheiten wie Graphiken, Dateien oder Screens zu manipulieren.

Dabei sorgt Blitz2 für die gesamte Speicherverwaltung der Objekte, einschließlich der Freigabe des Speichers, wenn das Programm beendet wird.

Obwohl die meisten Objekte spezielle Befehle benötigen, wird der Programmierer nie mit einer ungewöhnlichen Syntax konfrontiert. Alle Objekte können in der von Blitz2 gewohnten modularen Weise programmiert werden.

Die folgende Liste gibt einen Überblick über die wichtigsten Objekte, die Blitz2 kennt:

Files	für sequentiellen und direkten Zugriff auf DOS Dateien
Modules	Musikobjekte, die kompatibel zum Soundtracker sind
Blitzfonts	8x8 Pixel große Schriften für schnelle Textausgabe mit Bitmaps
IntuiFonts	skalierbare Schriften für Window-Ausgabe
Shapes	Standard Blitz2 Graphikelemente
Palettes	Farbpaletten
BitMaps	Standard Blitz2 Display-Elemente
Sounds	digitalisierte Soundbeispiele
Sprites	Hardware-Element für den Blitzmodus
Screens	Standard Intuition-Bildschirm
Windows	Standard Intuition-Fenster
Gadgets	Standard Intuition-Gadget
Menus	Standard Intuition-Menüs

Gemeinsamkeiten der Objekte

Für alle Blitz2-Objekte existiert ein spezieller Satz von Befehlen, mit denen sie definiert, manipuliert und natürlich auch zerstört werden können.

Den meisten Objekten ist ein eigenes Kapitel im Referenzhandbuch gewidmet, in dem alle Befehle ausführlich erklärt sind.

Alle Blitz2-Objekte können mit dem Befehl `Free` zerstört werden. Wurde ein Objekt bei Beendigung des Programms nicht gelöscht, so erledigt Blitz2 das automatisch.

Der Befehl `Free BitMap ()` gibt den für eine BitMap reservierten Speicherplatz frei. Dieser Befehl ist nützlich, wenn ein Objekt temporär benötigt wird und der Speicherplatz später im Programm noch gebraucht wird. Ansonsten kann man es Blitz2 überlassen, am Programmende aufzuräumen.

Maximalwerte

Die Anzahl der Objekte eines Typs, die vom Programm verwaltet werden können, ist begrenzt. Das Maximum läßt sich für jeden Objekt-Typ im Menü „Compiler Options“ angeben.

Wird versucht, mehr als die maximale Anzahl eines Objekt-Typs zu erzeugen, tritt der Laufzeitfehler „Value Out Of Maximum Range“ („Maximalwert überschritten“) auf.

Benutzung eines Objekts

Viele Befehle setzen voraus, das zuvor bestimmte Objekte erzeugt wurden. So muß z.B. vor der Ausführung des Befehls `Blit` (kopiert eine Shape auf eine Bitmap) sowohl ein Shape-Objekt als auch ein Bitmap-Objekt existieren.

Beim Aufruf des `Blit`-Befehls wird angegeben, welches Shape-Objekt „geblittet“ werden soll und Blitz2 kopiert die Shape auf die aktuelle Bitmap.

```
Use BitMap 0      ; Bitmap 0 zur aktuellen Bitmap machen
Blit 3,10,10      ; kopiere Shape 3 auf die aktuelle Bitmap
```

Durch den `Use`-Befehl wird die Bitmap 0 zur aktuellen Bitmap gemacht. Auch bei anderen Objekten, wie Screens, Windows und Paletten muß zunächst der `Use`-Befehl angewendet werden, bevor bestimmte Operationen ausgeführt werden können.

Durch die Erzeugung eines Objektes wird dieses automatisch zum aktuellen Objekt seiner Klasse.

Das Konzept der aktuellen Objekte wird in Blitz2 durchgängig genutzt. Es hat den Vorteil, daß die Programme schneller laufen und die Modularität der Programme erhöht wird.

Ein/Ausgabe-Objekte

Objekte vom Typ *BitMap*, *File* und *Window* sind Mittel zur Ein- und Ausgabe. Die Befehle `ObjektInput` und `ObjektOutput` ermöglichen es, Ein- und Ausgabekanäle umzuleiten.

Das `Print`-Kommando schreibt immer in das aktuelle Ausgabeobjekt, die Befehle `Edit` und `Inkey$` lesen immer vom gerade aktuellen Eingabeobjekt.

```
WindowOutput 2    ; Window 2 ist das aktuelle Ausgabeobjekt
Print "Hallo"
BitMapInput 1     ; Bitmap 1 wird zum aktuellen Eingabeobjekt
a$=Edit$(80)
```

Objekstrukturen (für Fortgeschrittene)

In Anhang 1 des Blitz2 Referenzhandbuchs befindet sich eine Beschreibung der inneren Struktur aller Objekte. Mit Hilfe des Befehls `Addr`, findet man die Speicherstelle an der eine Objekt-Struktur abgelegt ist.

Erfahrene Programmierer können mit Hilfe des `Addr`-Befehls, sowie mit den Befehlen `Peek` und `Poke` auf einzelne Elemente einer Objekstruktur zugreifen. Dies ist besonders bei bestimmten Systemobjekten, wie Screens und Windows nützlich, die Zeiger auf ihre Gegenstücke auf der Intuition-Seite enthalten.

Im folgenden Beispiel wird der Systemaufruf `ScreenToFront_` dazu verwendet, die Adresse der Intuition Screen aus der Struktur des Blitz2 Screen-Objekts zu ermitteln:

```
ScreenToFront_ Peek.l(Addr Screen(0))
```

Im nächsten Beispiel wird der Systemstruktur eines Windows ein Zeiger vom Typ `.Window` zugewiesen. Die *AmigaLibs.Res* muß als residente Datei geladen sein, wenn dieses Programm ausgeführt wird.

```
FindScreen 0
Window 0,10,10,100,100,9,"VERGROESSERE MICH",1,2
*w.Window=Peek.l(Addr Window(0))
WindowOutput 0
Repeat
      ev.l=WaitEvent
      WLocate 0,0
      NPrint *w\Width
      NPrint *w\Height
Until ev=$200
```

Beachten Sie den Unterschied zwischen `NewType .Window` und `NewType .window`:
`.Window` bezeichnet das System-Window (Intuition), während `.window` die Blitz2 Window-Struktur bezeichnet (s. Anhang 1 des Referenzhandbuchs).

Überblick über die primären Blitz2-Objekte

Screens
Screens können mit den Befehlen `Screen` und `FindScreen` erzeugt werden. Der Befehl `Screen` öffnet eine neue Screen, während `FindScreen` eine bereits vorhandene (meistens eine WorkBench Screen) zu einer Blitz2 Screen macht.

`FreeScreen` gibt eine Screen wieder frei, sollte aber erst dann aufgerufen werden, wenn alle Fenster zuvor geschlossen (also freigegeben) wurden.

Screen-Objekte erfüllen zwei Funktionen: sie bestimmen die Auflösung des Displays und der Palette und sie stellen die Fläche dar, auf der die Windows geöffnet werden. Beim Öffnen eines Windows oder bei der Benutzung von RGB- oder Paletten-Befehlen wird immer die aktuelle Screen benutzt.

Um beim Aufruf von Systemroutinen die Adresse der Systemstruktur .Screen zu ermitteln dient der Befehl `Peek.l(addr Screen(n))`.

Windows
Windows werden mit dem Befehl `Window` erzeugt. Gadgets oder Menüs werden immer zum aktuellen Window hinzugefügt. Zeichenbefehle wie `WPlot`, `WCircle`, `WLine` oder `WBox` beziehen sich ebenfalls immer auf das aktuelle Window.

Window-Objekte können Ein- und Ausgabe unter Verwendung der Befehle `WindowInput` und `WindowOutput` vornehmen, die Cursor-Position kann mit dem Befehl `WLocate` kontrolliert werden.

Bei der Freigabe von Windows braucht man sich nicht um die Freigabe der mit dem Window verbundenen Gadgets oder Menüs zu kümmern.

Gadget- und Menülisten
Gadgets und Menüs müssen vor ihrem Gebrauch zu Objekten zusammengefasst werden, die (na wie wohl ?) Gadgetlisten und Menülisten genannt werden. Diese Listen werden mit dem Window verbunden, wenn dieses erzeugt (geöffnet) wird. Alle Gadgets und Menüs müssen also bereits beim Programmstart in ihren Listen definiert sein.

Paletten
Paletten enthalten die RGB-Information (Rot, Grün, Blau) für alle darstellbaren Farben. Sie unterscheiden sich in folgender Weise von gewöhnlichen Blitz-Objekten:

Der Befehl `UsePalette` wendet die Farben, die in der angegebenen Palette definiert sind, auf die aktuelle Screen oder Slice an.

Der Befehl `RGB` sowie die Funktionen `Red()`, `Green()` und `Blue()` ändern die Farben der aktuellen Screen oder Slice, aber NICHT die der aktuellen Palette.

Es gibt keinen Befehl zur Erzeugung einer Palette. Paletten werden dadurch erzeugt, daß sie aus einer IFF-Datei geladen werden oder durch den Befehl `PalRGB`.

Bitmaps
Unter einer Bitmap versteht man ein Feld von Bildelementen (Pixels), aus denen sich das Display zusammensetzt.

Bitmaps können entweder mit dem Befehl `BitMap` erzeugt, von der Platte geladen oder von der Screen mit `ScreensBitMap` kopiert werden.

Bitmap-Objekte können mit `FreeBitMap` freigegeben werden, bis auf solche, die mit dem Befehl `ScreensBitMap` erzeugt wurden. Diese können nicht freigegeben werden.

Analog zu Windows, können auch Bitmaps zur Ein- und Ausgabe verwendet werden. Hierzu dienen die Befehle `BitMapInput` und `BitMapOutput`, die vorwiegend im *Blitzmodus* angewendet werden. Vor dem Aufruf von `BitMapInput` muß die Tastatur mit `BlitzKeys On` entsperrt werden.

Um den Cursor für die Ausgabe mit `BitMapOutput` zu positionieren, dient der Befehl `Locate`.

Shapes
Shapes sind Objekte, die Graphikelemente enthalten. Sie können entweder von der Platte geladen oder mit dem Befehl `GetAShape` aus einer Bitmap „ausgeschnitten“ werden.

Shapes werden mit dem allgemeinen `Free Shape n` Befehl freigegeben. Shapes, die in Verbindung mit Gadgets oder Menüs verwendet wurden, dürfen erst freigegeben werden, nachdem die entsprechenden Menü- und Gadgetlisten auch freigegeben wurden.

Shapes können auf vielfältige Weise manipuliert werden, sie können z.B. gedreht und skaliert werden.

Sprites
Sprites können initialisiert werden, indem sie von der Platte geladen werden oder indem ein Objekt vom Typ Shape in ein Objekt vom Typ Sprite umgewandelt wird. Die Shape kann anschließend freigegeben werden.

Der Befehl `FreeSprite n` gibt ein Sprite wieder frei.

Zur Zeit können Sprites lediglich im *Blitzmodus* verwendet werden. Im *Amigamodus* kann aber dennoch ein Zeiger auf ein Sprite Objekt definiert werden.

Slices
Eine Slice dient zur Konfiguration des Displays im *Blitzmodus*. Sie wird mit dem Befehl `Slice` erzeugt.

Im Gegensatz zu anderen Objekten, kann keine einzelne Slice freigegeben werden. Der Befehl `FreeSlices` gibt alle zu dem Zeitpunkt existierenden Slices frei.

Die Befehle `Show`, `ShowF`, `ShowB` und `ShowSprite` arbeiten alle mit der jeweils aktuellen Slice. Die Farbregister einer Slice werden von den Befehlen `RGB` und `UsePalette` beeinflußt.

Dateien
Dateien werden nicht wie andere Objekte erzeugt und freigegeben, sondern geöffnet und geschlossen.

Zum Öffnen dienen die Funktionen `OpenFile()`, `ReadFile()` und `WriteFile()`. Da bei Dateizugriffen Fehler auftreten können, werden hier Funktionen verwendet, sodaß erkannt werden kann, ob die Operation erfolgreich war.

Um eine Datei zu schließen, wird der Befehl `CloseFile n` verwendet, es gibt aber auch den Befehl `FreeFile n`. Es ist empfehlenswert, alle Dateien, die geöffnet wurden auch wieder „von Hand“ zu schließen und sich nicht darauf zu verlassen, daß Blitz2 dies beim Programmmende automatisch erledigt.

Natürlich ist eine Datei auch ein Ein/Ausgabe-Objekt. Die Befehle `FileInput` und `FileOutput` dienen dazu, die Ein/Ausgabe in eine Datei umzuleiten. Für die Befehle `Get`, `Put`, `ReadMem` und `WriteMem` ist dies jedoch nicht erforderlich, da sie mit einen Parameter vom Typ File# versorgt werden.

Zusammenfassung

Objekte in Blitz2 sind Datenstrukturen, die von den verschiedenen Bibliotheken verwendet werden und die eine Vielzahl von Einheiten behandeln. Blitz2 sorgt für die Speicherverwaltung der Objekte und gibt sie automatisch am Ende des Programms frei.

Viele der komplizierten Blitz2-Befehle können durch die Objekte auf einfachere Weise aufgerufen werden. Die Anzahl Parameter, die übergeben werden muß, kann durch das Konzept des „aktuellen Objekts“ minimiert werden.

Im Laufe der Zeit wird sich die Anzahl der verfügbaren Objekte in Blitz2 sicherlich noch erhöhen. Ebenso wird die Funktionalität der Objekte ständig erweitert.

Kapitel 8

Überblick über den Blitzmodus

- Der Blitzmodus
- Zaubereien mit Slices
- Der Blitter
- Der Copper
- Der QAmigamodus
- Zusammenfassung

Obwohl das Betriebssystem des Amiga allein schon sehr mächtig ist, nutzt es doch die Graphik-Fähigkeiten, die der Rechner besitzt, nicht aus. Der Blitzmodus ist für Programmierer wichtig, die reibungslose Animationen und Spiele o.ä. schreiben wollen.

Der Blitzmodus

Durch den Befehl `Blitz` geht das Programm in den *Blitzmodus* über. Im *Blitzmodus* ist das Betriebssystem „ausgeschaltet“, d.h es wird umgangen und das Programm hat die Kontrolle über den gesamten Rechner. Dies hat zur Folge, daß kein Multi-Tasking mehr läuft und keine Dateizugriffe mehr erfolgen können.

Der Vorteil des Blitzmodus liegt darin, daß die Programme wesentlich schneller ablaufen und dadurch der Bildschirm gleichmäßig gescrollt und ein sogenanntes „Dual-Playfield“ genutzt werden kann.

Der Blitzmodus ist nur ein temporärer Zustand. Kehrt das Programm in den *Amigamodus* zurück oder beendet es sich, erwacht das Betriebssystem wieder und übernimmt die Kontrolle über den Rechner.

Vorsicht ist geboten bei Verwendung des Blitzmodus im Zusammenhang mit der Version 1.3 oder früheren Versionen des Betriebssystems. Bei diesen Versionen kann das Wegschreiben (flush) der Dateipuffer bis zu zwei Sekunden dauern. Sie sollten daher absolut sicher sein, daß keine Plattenzugriffe mehr stattfinden, wenn Sie in den Blitzmodus übergehen. Zur Zeit ist uns keine Softwaremethode bekannt, mit der dies erreicht werden könnte. Das einzige, was wir empfehlen können, ist grundsätzlich den Befehl `VWAIT 100` abzusetzen, bevor in den *Blitzmodus* übergegangen wird.

Zaubereien mit Slices

Bei der Entwicklung des Amiga wurde sehr viel Wert auf die Graphikfähigkeiten gelegt. Im *Blitzmodus* wird die gesamte Graphikausgabe durch Slices übernommen. Diese sind wesentlich flexibler als die vom Betriebssystem verwendeten Screens und ermöglichen bestimmte Besonderheiten wie weiches (smooth) Scrolling, doppelt gepufferte Displays u.v.m.

Durch die Möglichkeit, mehr als eine Slice gleichzeitig auf dem Bildschirm zu haben, kann dieser in mehrere Abschnitte mit unterschiedlicher Auflösung zerlegt werden.

Es folgt eine Beschreibung der wichtigsten Vorteile der Darstellung mit Slices:

Weiches Scrollen
Das weiches Scrollen wird dadurch erreicht, daß nur ein Teil einer großen Bitmap auf dem Bildschirm angezeigt wird. Der Amiga gestattet es, das Anzeigefenster innerhalb einer großen Bitmap hin- und herzubewegen, wie die folgende Graphik zeigt:

>>> Diagram <<<

Das Display-Fenster repräsentiert den Bildschirmausschnitt, der jeweils zu sehen ist. Indem der Ausschnitt z.B. nach rechts über den Bildschirm bewegt wird, rollt das Bild gleichmäßig nach links.

Der Ausschnitt kann mit den Befehlen `Show`, `ShowF`, `ShowB` innerhalb der Bitmap bewegt werden.

Wie aus der obigen Darstellung ersichtlich, ist die Strecke, über die gescrollt werden kann, durch die Größe der Bitmap begrenzt. Wenn nun die linke Hälfte der Bitmap nochmals auf die rechte Seite kopiert wird, kann gleichmäßig nach rechts bis zum Ende gescrollt werden, bis das gleiche Bild wie ganz links erscheint. Ist das Ende erreicht, kann der Ausschnitt auf die linke Seite zurückgesetzt werden. Da der Ausschnitt an der linken und der rechten Seite der gleiche ist, merkt der Benutzer diesen Übergang nicht und der Vorgang erscheint wie ein endloses Scrollen.

>>> Diagram <<<

Dasselbe Vorgehen kann natürlich auch auf die vertikale Richtung angewendet werden.

Dual-Playfield
In bestimmten Situationen ist es erforderlich, einen Vordergrund unabhängig vom Hintergrund darzustellen. Der Amiga ermöglicht es, eine Bitmap über eine andere zu legen. Diese Darstellung wird Dual-Playfield Modus genannt.

Im Dual-Playfield Modus werden zwei Bitmaps mit je 8 Farben aufeinander dargestellt. Alle Pixel der Vordergrund-Bitmap, die die Farbe 0 besitzen, sind transparent, sodaß die Hintergrund-Bitmap durchscheint. Dabei können beide Ebenen unterschiedliche Farbdarstellungen besitzen.

>>> Diagram <<<

Der Copper

Eine gleichförmige Animation auf dem Bildschirm wird dadurch erreicht, daß die Graphik und die Video-Anzeige synchronisiert werden. Die Anzeige wird durch einen Video-Strahl erzeugt, der 50 mal in der Sekunde den Bildschirminhalt erneuert. Oft ist es sinnvoll, die Graphik mit dem vertikalen Signal zu koordinieren. Dies ermöglicht der Graphik-Koprozessor Copper.

Blitz2 bietet mehrere Möglichkeiten, den Copper auszunutzen. Eine beliebte Anwendung besteht darin, die Hintergrundfarbe so zu verändern, daß ein Regenbogeneffekt eintritt. Hierzu dient der Befehl `ColSplit`.

Für erfahrene Amiga-Programmierer bietet sich auch die Möglichkeit, weitere Effekte in vertikaler Richtung mit Hilfe des `CustomCop` Befehls zu programmieren.

Im Blitz2 Referenzhandbuch befinden sich Beispiele für diese Befehle.

Der Blitter

Der Amiga unterstützt die Darstellung von Graphiken in Form von Bitmaps durch eine spezielle Einrichtung, die Blitter genannt wird. Blitz2 enthält eine Reihe von Befehlen, mit denen Shapes auf Bitmaps ge-blittet werden können und außerdem ein besonderes Scroll-Kommando, mit denen Teile einer Bitmap, unter Zuhilfenahme des Blitters, bewegt werden können.

Es folgt eine kurze Übersicht über die verschiedenen Blitter-Befehle in Blitz2:

`Blit`	kopiert Shapes auf Bitmaps
`QBlit`	dasselbe wie `Blit` , Blitz2 merkt sich aber, an welche Stelle die Shape kopiert wurde und löscht sie wieder, wenn sie an eine andere Stelle bewegt werden soll.
`BBlit`	ähnlich wie `QBlit`, statt die Shape zu löschen, stellt Blitz2 jedoch den ursprünglichen Zustand der Bitmap vor dem `BBlit` wieder her.
`SBlit`	dasselbe wie `Blit`, aber mit einer Schablonen-Funktion die verhindert, daß bestimmte Bereiche der Bitmap überschrieben werden.
`Block`	eine schnelle Version von `Blit` , die jedoch nur mit rechteckigen Shapes arbeitet, die ein Vielfaches von 16 Pixeln groß sind.
`Scroll`	kopiert Teile einer Bitmap von einer Stelle zur anderen.

QAmiga-Modus

Mit Hilfe des `QAmiga` oder `Amiga` Befehls kann vom *Blitzmodus* zurück in den *Amigamodus* gesprungen werden.

Der `Amiga` Befehl stellt den Amiga-Anzeige wieder vollständig mit Mousepointer her.

Der `QAmiga` Befehl hingegen wechselt in den *Amigamodus*, ohne die Anzeige zu verändern. Hierdurch wird es möglich, in den *Amigamodus* zu wechseln, um bestimmte Operationen wie Dateizugriffe etc. durchzuführen und wieder zurück in den *Blitzmodus* zu wechseln, ohne den Bildschirm zu verändern.

Wichtiger Hinweis!!!
Bevor Sie in den *Blitzmodus* wechseln, müssen Sie absolut sicher sein, daß keinerlei Plattenzugriffe mehr stattfinden. Zur Zeit existiert keine Möglichkeit, dies mit Softwaremitteln sicherzustellen. Daher werden folgende Regeln für den *Blitzmodus* angegeben, die die Benutzung weitgehend sicher machen:

- Unbedingt auf das Erlöschen der Lampe für das Floppy-Laufwerk warten, bevor ein Programm gestartet wird, das sofort in den *Blitzmodus* wechselt.

- Benutzer von A590-Festplatten: immer auf das zweite Aufleuchten der Laufwerkslampe warten, wenn Sie Workbench 1.3 verwenden. Bei Workbench 2.0 werden alle Puffer sofort geleert.

- Wenn der `QAmiga` Befehl benutzt wird, um Daten auf die Platte zu schreiben, empfiehlt es sich, eine Verzögerung einzubauen, bevor zurück in den *Blitzmodus* gewechselt wird. Der Befehl `VWait 250` führt zu einer Verzögerung von etwa fünf Sekunden, das ist in jedem Fall ausreichend.

Es ist wichtig, sich zu vergegenwärtigen, daß keiner der Befehle, die auf das Betriebssystem angewiesen sind, im *Blitzmodus* verfügbar ist. Wird z.B. versucht, ein Window im *Blitzmodus* zu öffnen, wird das Compilieren mit der Meldung „Only available in Amiga Mode“ („Nur im Amigamodus verfügbar“) abgebrochen. Deshalb ist im Referenzhandbuch jeweils ausdrücklich angegeben, in welchem Modus die Befehle verfügbar sind.

Die Anweisungen `Blitz`, `Amiga` und `QAmiga` sind Compiler-Direktiven. Sie müssen daher in der korrekten Reihenfolge im Quellcode erscheinen.

Zusammenfassung

Blitz2 bietet Ihrem Programm zwei Umgebungen an. Der *Amigamodus* sollte für Anwendungsprogramme verwendet werden und ansonsten immer dann, wenn Plattenzugriffe erforderlich sind. Der *Blitzmodus* macht von den speziellen Graphikroutinen Gebrauch, die in Blitz2 implementiert sind. Diese ermöglichen eine Geschwindigkeit, die im *Amigamodus* einfach nicht erreicht werden kann. Dafür wird aber das Betriebssystem für die Zeit stillgelegt.

Abschließend läßt sich sagen, daß einzig und allein Unterhaltungsprogramme es sich leisten können, die Multitasking-Umgebung des Amiga auszuschalten. Reguläre Anwenderprogramme, die im *Blitzmodus* laufen, werden nicht gern gesehen.

Kapitel 9

Beispiele für den Blitzmodus

- Das Blitten von Teilfiguren
- Dual-Playfield Slice
- Doppelte Pufferung
- Weiches Scrollen

Das Blitten von Teilfiguren

Soll eine Shape auf eine Bitmap mit einem `Blit`-Befehl abgebildet werden, so muß die Shape vollständig auf die Bitmap passen. Ist dies nicht der Fall, tritt ein Fehler auf. Um eine Shape halb innerhalb und halb außerhalb einer Bitmap erschienen zu lassen, muß eine größere Bitmap verwendet und die Anzeige innerhalb der Bitmap positioniert werden. Die Größe des äußeren Rahmens hängt von der Größe der Shapes ab, die dargestellt werden sollen.

Im folgenden Beispiel wird eine Shape mit 32x32 Pixeln verwendet und infolgedessen werden weitere 32 Pixel um die Bitmap herum benötigt. Mit dem befehl `Show 0,32,32` wird das Bild in der vergrößerten Bitmap zentriert.

Außerdem wird die erweiterte Form des `Slice` Befehls verwendet, da eine Bitmap dargestellt wird, die größer als die Anzeige ist.

Die Funktion `RectsHit(x,y,1,1,0,0,320+32,256+32)` liefert den Wert *„wahr“* zurück, wenn sich die Shape innerhalb der vergrößerten Bitmap befindet und ge-blittet werden kann. Für eine größere Shape oder eine Shape mit zentrierter „Handle“, müssen die Parameter entsprechend verändert werden.

Die Routine `.makeshape` erzeugt eine temporäre Bitmap, die dazu benutzt wird, ein Muster zu generieren, das dann mit `GetaShape` in ein Shape-Objekt gewandelt wird.

```
BLITZ

Gosub makeshape

BitMap 0,320+64,256+64,3
Slice 0,44,320,256,$fff8,3,8,8,320+64,320+64
Show 0,32,32

While Joyb(0)=0
      x.w=Rnd(1024)-512
      y.w=Rnd(1024)-512
      If RectsHit(x,y,1,1,0,0,320+32,256+32)
            Blit 0,x,y
      EndIf
Wend


.makeshape:
      BitMap 1,32,32,3
      For i=1 To 15:Circle 16,16,i,i:Next
      GetaShape 0,0,0,32,32
      Free BitMap 1
```

```
        Return
```

Dual-Playfield Slice

Das folgende Programm demonstriert die Verwendung von Dual-Playfield Displays. Wie zuvor beschrieben wurde, dienen Dual-Playfields dazu, zwei Bitmaps gleichzeitig darzustellen. Hierzu dienen die beiden Befehle `ShowF` und `ShowB`.

Das Makro `rndpt` fügt einfach nur den Code `Rnd(640),Rnd(512)` in den Quellcode ein. So wird z.B. das Kommando `Line!rndpt,!rndpt,Rnd(7)` vom Compiler zu folgender Zeile erweitert:

```
Line Rnd(640),Rnd(512),Rnd(640),Rnd(512),Rnd(7)
```

Wiederum muß die erweiterte Form des `Slice` Befehls verwendet werden und die Flags auf $fffa gesetzt werden, um eine niedrigauflösende (lores), scrollbare Dual-Playfield Anzeige zu erhalten.

Man kann sich das Dual-Playfield als zwei getrennte Displays vorstellen: mit dem `ShowF`-Befehl wird der Vordergrund in Bitmap 1 positioniert, mit `ShowB` wird der Hintergrund in Bitmap 0 positioniert. Dabei muß die x-Position des jeweils anderen Displays an die Befehle `ShowF` und `ShowB` übergeben werden, sodaß Blitz2 intern die Positionen korrekt berechnen kann.

```
BLITZ

Macro rndpt Rnd(640),Rnd(512):End Macro

BitMap 0,640,512,3
For i=0 To 255
      Line !rndpt,!rndpt,Rnd(7)
Next

BitMap 1,640,512,3
For i=0 To 255
      Circlef !rndpt,Rnd(15),Rnd(7)
Next

Slice 0,44,320,256,$fffa,6,8,16,640,640

While Joyb(0)=0
      VWait
      x1=160+Sin(r)*160
      y1=128+Cos(r)*128
      x2=160-Sin(r)*160
      y2=128-Cos(r)*128
      ShowF 1,x1,y1,x2
      ShowB 0,x2,y2,x1
      r+.05
Wend
```

Doppelte Pufferung

Im folgenden Beispiel wird erläutert, wie ein doppelt gepuffertes Display verwendet wird, mit dem Graphiken

flimmerfrei bewegt werden können. Der Trick besteht darin, die eine Bitmap zu bearbeiten, während die andere Bitmap dargestellt wird, sodaß keine Störungen zu sehen sind.

Mit dem `VWait`-Befehl wird der Zeitpunkt abgepasst, an dem der vertikale Strahl oben am Bildschirm ist. Das ist der Moment, an dem die Bitmaps ausgetauscht werden müssen, ohne daß Störungen auftreten.

Mit der Anweisung `db=1-db` wird der Wert von `db` mit jeder Bildschirmfüllung zwischen 0 und 1 gewechselt. Die Bitmap `db` wird angezeigt mit `Show db, db` wird gewechselt (`db=1-db`) und anschließend wird mit `UseBitmap db` diese zur aktuellen Bitmap gemacht. Darin besteht die Methode der doppelten Pufferung, die eine Bitmap anzeigt, während die andere bearbeitet wird.

Da zwei Bitmaps verwendet werden, werden auch zwei Queues (Warteschlangen) benötigt, um `QBlit` verwenden zu können. Der Befehl `QBlit` führt ein normales Blit durch und speichert die Position des Objekts in einer Queue. Bei einem erneuten Aufruf von `QBlit` wird dann zunächst das Objekt von der vorherigen Position (die in der Queue gespeichert wurde) gelöscht und dann an der neuen Position dargestellt. Mit dem Befehl `UnQueue` können alle in der Queue gespeicherten Teile einer Screen gelöscht werden. So können in unserem Beispiel die Bälle in der neuen Position gezeichnet werden, ohne daß sie Schlieren von ihrer alten Position hinterlassen.

Mit der Anweisung `move#1,$dff180` wird die Hintergrundfarbe auf weiß gesetzt, sodaß deutlich wird, wieviel des Frames (Bildschirmfüllung) seit dem letzten VWait verstrichen ist, bedingt durch die Ausführung des Codes. Wenn die Anzahl der Bälle erhöht wird, so verlangsamt sich die Ausführung der Schleife (in der gezeichnet und positioniert wird) soweit, daß sie länger als ein Frame (1/50 Sekunde) dauert, und der Bildschirm beginnt zu blinken. In Kapitel 10 wird die Thematik der Frame-Raten noch näher erläutert.

Was noch zu erwähnen wäre, ist das Abprallen der Bälle, wenn sie an den Rand der Bitmap kommen. Die Richtung wird umgekehrt und zusätzlich wird die Richtung auch noch zu der Position der Bälle addiert, sodaß es nie dazu kommt, daß außerhalb der Bitmap ge-blittet wird.

```
BLITZ

n=25

NEWTYPE .ball
      x.w:y:xa:ya
End NEWTYPE

Dim List b.ball(n-1)
While AddItem(b())
      b()\x=Rnd(320-32),Rnd(256-32),Rnd(4)-2,Rnd(4)-2
Wend

Gosub getshape

BitMap 0,320,256,3
BitMap 1,320,256,3
Queue 0,n
Queue 1,n
Slice 0,44,3

While Joyb(0)=0
      VWait
      Show db
      db=1-db
      Use BitMap db
      UnQueue db
      ResetList b()
      USEPATH b()
```

```
      While NextItem(b())
            \x+\xa:\y+\ya
            If NOT RectsHit(\x,\y,1,1,0,0,320-32,256-32)
                  \xa=-\xa:\ya=-\ya
                  \x+\xa:\y+\ya
            EndIf
            QBlit db,0,\x,\y
      Wend
      MOVE #-1,$dff180
Wend

End

.getshape:
      BitMap 1,32,32,3
      For i=1 To 15:Circle 16,16,i,i:Next
      GetaShape 0,0,0,32,32
      Free BitMap 1
      Return
```

Weiches Scrollen

In diesem letzten Beispiel wird das weiches Scrollen erläutert, wie es im letzten Kapitel beschrieben wurde.

Mit dem Befehl `Scroll` wird die linke Seite der Bitmap nach rechts und die obere Hälfte nach unten kopiert. Dadurch wird die Bitmap vierfach so groß und enthält das selbe Bild in jedem Quadranten.

Dies ermöglicht es, die Bitmap durch den Bildschirm scrollen zu lassen und wenn der rechte Rand erreicht ist, wieder zum linken Rand zurück zu springen, ohne daß ein Bruch sichtbar wird, da das Bild links und rechts gleich ist. Das gleiche gilt für das auf- und abscrollen.

Um die Maus-Bewegungen des Benutzers zu ermitteln, muß zunächst der Befehl `Mouse On` abgesetzt werden. Anschließend wird ermittelt, um welchen Betrag sich die Maus bewegt hat, und dieser Wert wird zu der Geschwindigkeit hinzuaddiert, mit der das Display verschoben wird.

Mit dem Befehl `QLimit(xa+MouseXSpeed,-20,20)` wird sichergestellt, daß die Variable `xa` (x_add) immer im gültigen Bereich von -20 bis 20 bleibt.

Der Übergang von einem zum anderen Rand wird durch die Anweisung `x=QWrap(xa+MouseSpeed,-20,20)` erreicht.

```
BLITZ
Mouse On
n=25
BitMap 0,640,512,3

For i=0 To 150
      Circlef Rnd(320-32)+16,Rnd(256-32)+16,Rnd(16),Rnd(8)
Next

Scroll 0,0,320,256,320,0
Scroll 0,0,640,256,0,256

Slice 0,44,320,256,$fff8,3,8,8,640,640
```

```
While Joyb(0)=0
      VWait
      Show db,x,y
      xa=QLimit(xa+MouseXSpeed,-20,20)
      ya=QLimit(ya+MouseYSpeed,-20,20)
      x=QWrap(x+xa,0,320)
      y=QWrap(y+ya,0,256)
Wend
```

Kapitel 10

Weiterführende Themen

- Residente Dateien
- Betriebssystem-Aufrufe
- Auffinden von Variablen und Labels im Speicher
- Konstanten
- Bedingte Compilierung
- Makros
- Inline Assembler

Residente Dateien

Programme, die mit einer großen Anzahl von NewTypes, Makros oder Konstanten arbeiten, können einfacher geschrieben werden, in dem sogenannte residente Dateien verwendet werden.

Eine residente Datei enthält alle NewTypes, Makros und Konstanten in vor-compilierter, d.h. binärer Form. Dadurch können alle diese Definitionen aus dem eigentlichen Programmcode weggelassen werden, wodurch der Code kleiner wird und sich schneller compilieren läßt.

Um eine solche residente Datei zu erzeugen, müssen alle NewType-Definitionen, Makros und Konstanten in einem Programm zusammengefasst werden. Dieses Programm wird dann in residente Form übersetzt. Es folgt ein Beispiel für ein solches Programm:

```
NEWTYPE.test
      a.l
      b.w
End NEWTYPE

Macro mac
      NPrint "Hallo"
End Macro

#const=10
```

Jetzt muß das Programm lediglich mit „COMPILE & RUN“ compiliert werden und anschließend mit „CREATE RESIDENT“ (Compiler-Menü) in eine residente Datei gewandelt werden. Es erscheint dann ein File-Requester, der Sie auffordert, einen Namen für die Datei einzugeben. Mehr muß nicht gemacht werden.

Eine residente Datei kann in ein beliebiges Programm eingebunden werden, indem der entsprechende Dateiname in eines der RESIDENT-Felder des Compiler Options Menüs eingetragen wird. Damit sind alle in der residenten Datei enthaltenen NewTypes, Makros und Konstanten automatisch für das Programm verfügbar.

Die mitgelieferte Datei *AMIGALIBS.RES* enthält alle Strukturen und Makros, die für den Umgang mit dem Betriebssystem erforderlich sind. Wer schon einmal nahe am Betriebssystem programmiert hat, wird es zu schätzen wissen, daß das übliche Einlesen der Header-Dateien entfällt und damit Minuten beim Compilieren gespart werden.

Betriebssystem-Aufrufe

Es wurde bei der Entwicklung von Blitz2 viel Mühe darauf verwendet, dem Programmierer möglichst weitgehenden Zugriff auf das mächtige Amiga-Betriebssystem zu gewähren.

Der Aufruf von Betriebssystem-Bibliotheken
Nicht alle Befehle des Betriebssystem werden von den internen Blitz2-Befehlen abgedeckt. Dennoch stehen dem erfahrenen Programmierer alle Routinen der Exec-, Intuition-, DOS-, und Graphics-Libraries durch Blitz2 zur Verfügung (s. Anhang 5 des Blitz2 Referenzhandbuchs).

Der Zugriff auf die Standard-Libraries ist durch das „Blitz2 Advanced Programmers Pack“ von Acid Software gewährleistet.

Im folgenden Beispiel werden Routinen der Amiga ROM-Graphics und der Intuition Libraries gerufen.

```
FindScreen 0                        ;workbench screen verwenden

;Window oeffnen

Window 0,0,10,320,180,$408,"",1,2
rp.l=RastPort(0)                    ;Rastport fuer window ermitteln
win.l=Peek.l(Addr Window(0))        ;Window-Struktur ermitteln

DrawEllipse_ rp,100,100,50,50       ;graphics library
MoveWindow_ win,8,0                 ;intuition library
BitMap 1,320,200,2                  ;Arbeitsbitmap erstellen
Circlef 160,100,100,1               ;etwas zeichnen

;Jetzt in das Window transferieren

BltBitMapRastPort_ Addr BitMap(1),0,0,rp,0,0,100,100,$c0

WaitEvent
```

Mit dem letzten Befehl `BltBitMapRastPort_` kann eine Graphik, die mit den schnelleren Blitz2 Bitmap-Befehlen erzeugt wurde, in einem Window ausgegeben werden. Dadurch können die Blitz2-Befehle noch universeller angewendet werden.

Zugriff auf Strukturen des Betriebssystems
Wenn die residente Datei *AMIGALIBS.RES* vorhanden ist (s. Anfang des Kapitels), ist sogar ein noch weitergehenderer Zugriff auf das Betriebssystem möglich. Im folgenden Beispiel wird auf Strukturen des Betriebssystems zugegriffen:

```
;Variable *exec zeigt auf die ExecBase Struktur
;Variable *mylist zeigt auf den Typ List
;Variable *mynode zeigt auf den System Knoten

*exec.ExecBase=Peek.l(4)
*mylist.List=*exec\LibList
*mynode.Node=*mylist\lh_Head

While *mynode\ln_Succ
      a$=Peek$(*mynode\ln_Name)     ;Knotennamen drucken
      NPrint a$
      *mynode=*mynode\ln_Succ       ;gehe zum naechsten Knoten
Wend
```

```
MouseWait
```

Durch die Angabe des Sterns in `*variablenname.typ` wird Blitz2 angewiesen, statt einer Variablen dieses Typs lediglich einen Zeiger (Pointer) auf eine solche Variable anzulegen. Anschließend kann die Variable (Struktur) behandelt werden wie eine interne Blitz2-Variable.

Mit dem Befehl `Peek$` wird ein String aus einer Betriebssystem-Struktur direkt in eine String-Variable eingelesen, bis eine Null (`CHR$(0)`) gelesen wurde.

Das Auffinden von Variablen und Labels im Speicher

Mit dem kommerziellen Und-Zeichen („&“, engl. „ampersand“) wird die Adresse einer Variablen im Speicher referenziert:

```
;
; Ein Beispiel fuer '&' um die Adresse einer Var. zu finden
;
Var.l=5
Poke.l &Var,10
NPrint Var
MouseWait
```

Diese Methode entspricht der Funktion *VarPtr* in anderen BASIC-Versionen.

Wird die Adresse einer String-Variablen ermittelt, so zeigt der Pointer auf das erste Zeichen des Strings. Die Länge des Strings ist in einem Doppelwort an „Adresse-4“ abgespeichert.

Das Fragezeichen „?“ dient zum Auffinden der Adresse einer Sprungmarke (Label) des Programms im Speicher. Zum Beispiel so:

```
;
; Ein Beispiel das die Adresse eines Programms findet
;
MOVE #10,There          ;Sogar Assembler-Code hier
NPrint Peek.w(?There)
MouseWait
End
;
There:Dc.w 0            ;und hier auch
```

Die oben beschriebenen Methoden sind im Grunde nur für Programmierer mit Assembler-Erfahrung von Interesse, die ihre Ziele auf unkonventionelle Weise verfolgen.

Konstanten

Unter einer Konstanten versteht man in BASIC einen Wert, der sich während der gesamten Ausführung des Programms nicht ändert. Die Zahl 5 in der Anweisung `a=5` z.B. ist eine Konstante.

Das „Lattenkreuz“ „#“ (engl. „Hash sign“) vor einem Variablennamen kennzeichnet diese als eine Konstante, d.h. ihr Wert kann sich im Laufe des Programms nicht mehr ändern. Die Anweisung `#weite=320` legt die

Variable `#weite` endgültig auf den Wert 320 fest.

Konstanten besitzen bestimmte Eigenschaften, sie:

- sind schneller als Variablen und verbrauchen keinen Speicherplatz
- erhöhen die Lesbarkeit des Programmcodes
- können ebenfalls im Assembler verwendet werden
- können für die Auswertung von bedingten Compilier-Anweisungen verwendet werden
- dürfen nur Integerwerte enthalten
- vereinfachen die Verwendung eines konstanten Wertes, der an verschiedenen Stellen im Programm verwendet wird
- können im Quellcode zum Zeitpunkt der Compilierung verändert werden, aber nicht mehr danach während der Laufzeit.

Der wichtigste Aspekt bei der Verwendung von Konstanten, vom Standpunkt des Programmierers aus, ist die Tatsache, daß bestimmte „Naturkonstanten“, die an unterschiedlichen Stellen des Programms benötigt werden, durch aussagefähige Namen ersetzt werden, wie z.B. `#weite`.

Sollte der Wert von `weite` jemals geändert werden müssen, braucht dann lediglich an einer Stelle die Zuweisung `#weite=320` in z.B. `#weite=640` geändert zu werden, anstatt den gesamten Programmcode nach der Zahl 320 abzusuchen.

Bedingte Compilierung

Die bedingte Compilierung gestattet es, den Compiler quasi ein- und auszuschalten, während dieser sich durch den Programmcode arbeitet.Dadurch ist zu kontrollieren, welche Teile des Programmcodes übersetzt werden und welche nicht.

Dies kann dazu gebraucht werden, um unterschiedliche Versionen eines Programms zu erzeugen, ohne zwei Versionen des Quellcodes pflegen zu müssen. Außerdem kann damit z.B. eine abgemagerte Demo-Version eines Programms erzeugt oder das Programm auf unterschiedliche Hardware-Konfiguration eingestellt werden.

Ebenso kann das bedingte Übersetzen bei der Fehlersuche hilfreich sein, indem - für diesen Zweck - unwichtige Teile des Codes ausgeschaltet werden oder zusätzliche Überprüfungen eingebaut werden. Wir hoffen allerdings, daß der Blitz2 Debugger dies nicht erforderlich macht.

Folgende Compiler-Direktiven stehen zur Verfügung:

`CNIF`	Compiler einschalten, wenn der numerische Vergleich wahr ist
`CSIF`	Compiler einschalten, wenn der String-Vergleich wahr ist
`CELSE`	schaltet den Compiler in den entgegengesetzten Zustand, also ein=>aus und aus=>ein
`CEND`	Ende eines Blocks der bedingt übersetzt werden soll, der vorherige Zustand des Compilers wird wiederhergestellt

Der Compiler besitzt einen internen Ein/Aus-Schalter. Dieser wird durch die Anweisungen `CNIF` und `CSIF` ein- oder ausgeschaltet, `CELSE` wechselt den Zustand des Schalters und `CEND` stellt den Zustand vor der letzten `CNIF`- oder `CSIF`-Anweisung wieder her. `CNIF`/`CEND`-Blöcke können auch verschachtelt werden.

Da die Compiler-Direktiven bereits beim Übersetzen ausgewertet werden, können in den Anweisungen `CNIF` und `CSIF` nur Konstanten getestet werden, also „5“ oder „#test“. Der Wert von Variablen ist hingegen zur Compilezeit nicht bekannt, sondern erst bei der Ausführung des Programms, daher können Variablen nicht in

Verbindung mit diesen Anweisungen verwendet werden.

Im folgenden Beispiel wird der Gebrauch der bedingten Compilierung erläutert:

```
#demover=1  ;dies ist eine Demoversion

NPrint "Goo Goo Software (c)1993"

CNIF #demover=1
      NPrint "DEMONSTRATIONS-VERSION"
CELSE
      NPrint "REGISTRIERTE VERSION"
CEND
;
; und spaeter im Programm...
;
.SaveRoutine
CNIF #demover=0   ;nur wenn keine Demoversion
                  ;
                  ;Save-Routine ausfuehren
                  ;
CEND
Return
```

Der Vorteil dieses Vorgehens gegenüber einer einfachen `If demover=0...EndIf`-Konstruktion besteht darin, daß hierbei der Code für die Abspeicher-Routine überhaupt nicht im Programm enthalten ist und daher auch nicht von Hackern entschlüsselt werden kann.

Die entscheidenden Vorteile des bedingten Compilierens kommen jedoch erst bei der Makro-Programmierung zum tragen.

Makros

Makros werden vorwiegend bei der Assembler-Programmierung oder in maschinenorientierten Sprachen verwendet. Sie werden zu verschiedenen Zwecken verwendet: um Schreibarbeit zu sparen, um einfache Routinen durch schnellere „Inline“-Versionen zu ersetzen, oder auch um Code zu erzeugen, der in Hochsprachen nicht herzustellen ist.

Ein Makro wird durch die Befehlsfolge `Macro name ... End Macro` definiert. Der Code, der zwischen `Macro` und `EndMacro` steht, wird im Gegensatz zu normalem Programmcode nicht übersetzt, sondern zunächst vom Compiler gespeichert. Wird nun ein Makro mit `!macroname` aufgerufen, ersetzt der Compiler den Makronamen mit dem Inhalt des entsprechenden Makros im Quellcode.

Das folgende Beispiel

```
Macro mymacro
      a=a+1
      NPrint "Viel Glueck"
End Macro

NPrint "Dummes Beispiel v1.0"
!mymacro
!mymacro
MouseWait
```

wird vom Compiler erweitert, sodaß sich ergibt:

```
NPrint "Dummes Beispiel v1.0"
a=a+1
NPrint "Viel Glueck"
a=a+1
NPrint "Viel Glueck"
MouseWait
```

Parameterübergabe bei Makros
Es können auch Parameter an ein Makro übergeben werden, dadurch werden sie noch nützlicher. Parameter werden in geschweiften Klammern *{* und *}* eingeschlossen. Bei der Auswertung des Makros werden zunächst die Parameter in den Text des Makros eingefügt und dann das Makro ersetzt.

In der Definition eines Makros wird die Stelle, an der der Parameter eingefügt werden soll, mit dem umgekehrten Apostroph (auf der Amiga-Tastatur oberhalb der *TAB*-Taste) gefolgt von einer Zahl oder einem Buchstaben (1-9 oder a-z) markiert.

Das folgende Beispiel erläutert die Übergabe von zwei Parametern an ein Makro:

```
Macro distance
      Sqr(`1*`1+`2*`2)
End Macro

NPrint !distance{20,30}

MouseWait
```

Der Compiler ersetzt nun jedes Auftreten vom `` `1 `` durch den ersten und `` `2 `` durch den zweiten Parameter.

```
NPrint Sqr(20*20+30*30)
```

Wenn mehr als 9 Parameter übergeben werden sollen, werden Buchstaben verwendet: `` `a `` stellt den zehnten, `` `b `` den elften Parameter dar usw.

Beim Aufruf können die Parameter aus beliebigem Text bestehen, statt `{20,30}` hätte im obigen Beispiel auch `{x,y}` stehen können.

Achtung: Bei der Übergabe von mathematischen Ausdrücken als Parameter an Makros muß unbedingt darauf geachtet werden, daß diese korrekt geklammert sind! Es treten sonst ungewollte Seiteneffekte auf, die sehr schwer zu ermitteln sind, wie hier:

```
!distance{x*10+20,(y*10+20)}
```

Diese Zeile wird zu folgendem Ausdruck expandiert:

```
Sqr(x*10+20*x*10+20+(y*10+20)*(y*10+20)}
```

Da in dem Makro die beiden Parameter jeweils mit sich selbst multipliziert werden, wird der erste Parameter nicht richtig ersetzt. Dadurch, daß die Klammern fehlen, wird die Multiplikation falsch ausgeführt. Der zweite Parameter hingegen wird korrekt ausgewertet, da der Ausdruck in Klammern angegeben ist.

Der `` `0 ``-Parameter
Der Parameter mit der Nummer `` `0 `` besitzt eine spezielle Bedeutung: er gibt die Anzahl der nachfolgenden Parameter an. Dies kann sowohl zur Überprüfung der korrekten Anzahl der Parameter verwendet werden, als auch dafür, bewußt eine variable Anzahl von Parametern an das Makro zu übergeben.

Das folgende Makro prüft, ob genau zwei Parameter übergeben wurden und erzeugt gegebenenfalls eine

Compiler-Meldung:

```
Macro Vadd
CNIF `0=2
      `1=`1+`2
CELSE
      CERR "Falsche Anzahl Parameter in '!Vadd'"
CEND
End Macro

!Vadd{a}
```

Wenn Sie dieses Programm übersetzen lassen, gibt der Compiler die entsprechende Meldung aus, sobald die Stelle `!Vadd{a}` erreicht ist. Die Anweisung `CERR` ist eine spezielle Compiler-Direktive, die zur Erzeugung von Anwender-spezifischen Fehlermeldungen dient.

Rekursive Makros
Makros können auch rekursiv programmiert werden, d.h. sie können sich selbst aufrufen. Im folgenden Beispiel druckt das Makro den ersten Parameter und ruft sich dann anschließend selbst ohne den ersten Parameter auf. Dadurch wird praktisch die Liste der Parameter nacheinander abgearbeitet, bis das Null-Zeichen, d.h. kein Parameter erreicht ist.

```
Macro dolist      ;bis zu 16 Variablen auflisten
      NPrint `1
      CSIF "`2">""
      !dolist{`2,`3,`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e,`f,`g}
      CEND
End Macro
!dolist {a,b,c,d,e,f,g,h,i}
MouseWait
```

Funktionen durch Makros ersetzen
Makros eignen sich besonders dafür, Funktionen zu ersetzen, die keine lokalen Variablen benötigen, dafür aber mehr als einen Wert zurückliefern sollen. Im folgenden Makro `project` werden die Parameter `x`, `y` und `z` auf eine zweidimensionale x-y-Ebene projiziert. Das Ergebnis kann dann wiederum als Parameter für Zeichenbefehle verwendet werden.

```
Macro project #xm+`1*9-`2*6,#ym+`1*3+`2*6-`3*7:End Macro

#xm=320:#ym=256

Screen 0,28:ScreensBitMap 0,0

For z=-15 To 15
      For y=-15 To 15
            For x=-15 To 15
                  Circlef !project{x,y,z},3,x&y&z
            Next
      Next
Next

MouseWait
```

Das CMake-Zeichen
Das Zeichen „~“ hat eine besondere Bedeutung: es dient dazu, das Ergebnis eines Ausdrucks als Literal (d.h. nicht als Wert, sondern als Zeichen) wiederum in den Code einzufügen. Dies kann sehr nützlich sein, wenn z.B. Variablennamen oder Labels erzeugt werden sollen, die sich zum einen Teil aus einem Parameter

und zum anderen Teil aus einem konstanten Ausdruck zusammensetzen.

```
var2=20
var3=30

Macro lvar
      NPrint var~`1~
End Macro

!lvar{2+1}

MouseWait
```

In diesem Beispiel würde ohne das CMake-Zeichen der Wert 21 ausgegeben werden, da der Ausdruck `var`1` mit dem Parameter `2+1` zu `var2+1` ersetzt werden würde. Durch das CMake-Zeichen wird nun aber der als Parameter übergebene Ausdruck zunächst ausgewertet und erst dann erfolgt die Ersetzung: es ergibt sich `var3`.

Inline Assembler

Es besteht die Möglichkeit, Maschinenbefehle des 68000 innerhalb von Blitz2-Programmen mit Hilfe des Inline-Assemblers aufzurufen. Dadurch können erfahrene Programmierer ihre Programme beschleunigen, indem sie bestimmte Blitz2-Routinen durch die entsprechenden schnelleren Maschinenbefehle ersetzen.

Es gibt drei Methoden, Assembler in Blitz2 zu integrieren:

1. zeilenweise auf Variablen mit den Befehlen `GetReg` und `PutReg` zugreifen
2. ganze Befehle und Funktionen schreiben
3. eigene Blitz2-Libraries erzeugen

GetReg & PutReg
Mit den Befehlen `GetReg` und `PutReg` kann auf die BASIC-Variablen zugegriffen werden. Im folgenden Beispiel wird die Benutzung dieser beiden Befehle erläutert:

```
a.w=5             ;Word-Variablen benutzen
b.w=10
GetReg d0,a       ;Wert von a=>d0
GetReg d1,b       ;Wert von b=>d1
MULU d0,d1
PutReg d1,c.w     ;Wert von d1=>c
NPrint c
MouseWait
```

Im nächsten Beispiel wird die erste Bitplane der Bitmap 0 invertiert. Dabei kann ein beliebiger Ausdruck zusammen mit dem `GetReg`-Befehl verwendet werden. Da `GetReg` nur mit Datenregistern arbeiten kann, muß die Bitmap-Struktur zunächst in d0 angelegt und dann nach a0 verschoben werden.

```
Screen 0,3
ScreensBitMap 0,0
While Joyb(0)=0
      VWait 15
      Gosub inverse
Wend
End
```

```
inverse:          ;Speicheradress der Bitmap-Struktur=>d0
      GetReg d0,Addr BitMap(0)
      MOVE.l d0,a0
      MOVEM (a0),d0-d1
      MULU d0,d1
      LSR.l#2,d1
      SUBQ#1,d1
      MOVE.l 8(a0),a0
loop:
      NOT.l (a0)+
      DBRA  d1,loop
      Return
```

Prozeduren in Assembler
Noch effizienter ist es, ganze Funktionen und Befehle als Assembler-Routinen zu schreiben. Die Parameter werden automatisch in den Registern d0-d5 abgelegt und bei Funktionen wird der Wert in d0 zurückgegeben.

Da das Adress-Register a4 die Basis der lokalen Variablen darstellt, muß zu Beginn der Prozedur der Befehl `UNLK a4` stehen. Weiterhin muß die Prozedur hundertprozentig aus Assembler-Befehlen bestehen und die Adress-Register a4 bis a6 dürfen nicht verändert werden.

Im folgenden Beispiel wird gezeigt, wie der Befehl `qplot{}` in Assembler realisiert und von einer BASIC-Routine aufgerufen wird. `Qplot` setzt ein Pixel auf der ersten Bitplane der angegebenen Bitmap. Wie man sieht, können mehrere Assembler-Befehle auf einer Codezeile stehen.

```
Statement qplot{bmap.l,x.w,y.w}
      UNLK a4
      MOVE.l d0,a0:MULU (a0),d2
      MOVE.l 8(a0),a0:ADD.l d2,a0
      MOVE d1,d2:LSR#3,d2:ADD d2,a0:BSET.b d1,(a0)
      RTS
End Statement

Screen 0,1
ScreensBitMap 0,0
bp.l=Addr BitMap(0)
For y.w=0 To 199
      For x.w=0 To 319
            qplot{bp,x,y}
      Next
Next
MouseWait
```

Es können auch ganze Bibliotheken mit Routinen in Machinensprache angelegt werden. Dies ist mit dem „Advanced Programmers Pack“ von Acid Software möglich, das ein sehr mächtiges Library-System enthält. Es stellt eine effektive Entwicklungsumgebung für die fortgeschrittene Programmentwicklung zur Verfügung.

Kapitel 11

Display-Library & AGA

Einführung

Die Display-Library ist eine wichtige Erweiterung für Blitzbasic2. Entwickelt als Ersatz für Slices bietet sie nicht nur Spieleprogrammierern Zugriff auf das AGA Chipset und einen modularen Ansatz zur Steuerung der Amiga Grafikhardware.

Die Anzeige des Amiga wird durch den Koprozessor Copper gesteuert, dieser führt in jedem Frame eine Liste von Anweisungen aus. Also 50 Mal pro Sekunde auf PAL Amigas. Der Kathodenstrahl der Zeile für Zeile über den Bildschirm geht und jedes einzelne Pixel zeichnet wird durch eine Reihe von Hardware-Registern gesteuert. Der Copper hat hierbei die Aufgabe alles synchron zu halten.

Eine Copperlist beinhaltet Informationen über die Farben, Bitplanes, Sprites, die aktuelle Auflösung und vielem mehr. Diese Informationen benötigt der Kathodenstrahl (Video-Strahl) um die Anzeige zu rendern.

Initialisierung

Im Gegensatz zu Slices, welche angezeigt werden solbald sie initialisiert werden, benötigt die Display-Library sogenannte Copperlisten zur Initialisierung. Diese Copperlisten werden mit `InitCopList` erstellt. Die Funktion `CreateDisplay` erzeugt ebenfalls eine Copperliste. Ein wichtiger Unterschied zu Slices ist dass diese jedesmal Speicher allozieren wenn eine Änderung der Anzeige erforderlich ist, während die Display-Library mehrere Copperlisten initialisieren kann bevor das Display erzeugt wird.

Es gibt zwei Formen von `InitCopList`. Die kurze Variante erfordert lediglich die Copperlisten-Nummer und die entsprechenden Flags. Die Höhe des Displays ist dann auf 256 Pixel voreingestellt. Die Breite des Displays kann 320, 640 oder 1280 sein, je nachdem welche Auflösung und Farbtiefe man mit den Flags definiert hat.

Die längere Version von `InitCopList` hat folgendes Format:

```
InitCopList CopList#, ypos, height, type, sprites, colors, customs
```

Der Parameter `ypos` wird in der Regel auf 44 gesetzt was der Oberkante eines PAL-Bildschirms entspricht. Wenn die Copperliste unterhalb einer anderen Copperliste benutzt wird, dann muss `ypos` 2 Zeilen tiefer liegen als die letzte Zeile der vorher gehenden Copperliste.

Der Parameter `sprite` sollte immer auf 8 gesetzt werden auch wenn diese Menge nicht erforderlich ist. Der Parameter `colors` wird auf die gewünschte Zahl gesetzt. Wenn mehr als 32 Farben verwendet werden sollen muss immer der Flag `#agacolors` gesetzt werden!

Der Parameter `customs` bietet Raum für erweiterte Copperlisten die jedem Display zugewiesen werden können. Die Verwendung von `custom` wird später in diesem Kapitel besprochen.

Flags

Der Parameter Flags berechnet sich durch die Addition der einzelnen Werte. Beachte: Wenn dem Parameter `flags` eine Variable zugewiesen soll in der zuvor die Flags abgelegt worden sind, dann muss diese Variable vom Typ *Long* (.l = 32Bit) sein!

`#onebitplane`	`$01`	
`#twobitplanes`	`$02`	
`#threebitplanes`	`$03`	
`#fourbitplanes`	`$04`	
`#fivebitplanes`	`$05`	
`#sixbitplanes`	`$06`	
`#sevenbitplanes`	`$07`	nur für AGA
`#eightbitplanes`	`$08`	nur für AGA
`#smoothscrolling`	`$10`	Setzen wenn Du Bitmaps scrollen willst.
`#dualplayfields`	`$20`	Ermöglicht den Dual Playfield Modus
`#extrahalfbright`	`$40`	Erzwingt den EHB Modus mit 6 Bitplanes
`#ham`	`$80`	HAM Modus
`#lores`	`$000`	
`#hires`	`$100`	
`#superhires`	`$200`	
`#loressprites`	`$400`	
`#hiressprites`	`$800`	nur für AGA
`#superhiressprites`	`$c00`	nur für AGA
`#fetchmode0`	`$0000`	
`#fetchmode1`	`$1000`	nur für AGA
`#fetchmode2`	`$2000`	nur für AGA
`#fetchmode3`	`$3000`	nur für AGA
`#agacolors`	`$10000`	nur für AGA

Der Flag `#agacolors` muss stets gesetzt werden wenn mehr als 32 Farben oder 24Bit Farbdefinitionen verwendet werden sollen.

Smoothscrolling – weiches Scrollen

Wenn das Flag `#smoothscrolling` gesetzt wurde muss die erweiterte Form der Funktion `DisplayBitmap` verwendet werden damit die Bitmap an jeder möglichen Position angezeigt werden kann, das ermöglicht es dem Programmierer die Bitmap zu scrollen die gerade angezeigt wird.

Beachte:

- Benutze immer die erweiterte Version von `DisplayBitmap` bei gesetztem `#smoothscrolling` Flag, auch wenn die Bitmap an Koordinate 0, 0 angezeigt wird.
- `DisplayBitmap` akzeptiert für die X-Koordinate Werte vom Typ *Quick* (.q) und positioniert die Bitmap auf AGA Maschinen in Pixel-Gruppen.
- Die Breite des Displays verringert sich gegenüber den Standardwerten 320, 640, 1280 wenn `#smoothscrolling` aktiviert ist.

Dualplayfields

Wenn das Flag `#dualplayfields` gesetzt ist werden zwei Bitmaps übereinander auf dem selben Display angezeigt. Eine Kombination der Flags `#dualplayfields` und `#smoothscrolling` gestattet Parallax-Effekte. Dieser Effekt tritt auf wenn beide Bitplanes unterschiedlich schnell gescrollt werden. Beachte dass es auf AGA Maschinen möglich ist zwei 16 farbige Bitmaps zu verwenden wenn `#dualplayfields` aktiviert ist und die Anzahl der Bitplanes auf 8 gesetzt wird.

Sprites

Die Anzahl der verfügbaren Sprites hängt von der Art der Anzeige und den Fetchmode Einstellungen ab. Die meisten AGA Modis erfordern dass das Display hoizontal verkleinert wird um 8 Sprites gleichzeitig darstellen zu können. Aktuell kann dies nur durch die Verwendung der Funktion `DisplayAdjust` erreicht werden. Beispiele hierzu sind auf der „Blitz Examples“ Diskette zu finden.

Die AGA Hardware ermöglicht dem Programmierer die Nutzung von LowRes, HighRes und SuperHighRes Sprites. Die höheren Auflösungen ermöglichen es einem Grafiker Dithering (der gezielte Einsatz von Pixel-Rauschen um harte Farbübergänge zu vermeiden) einzusetzen. Dies ist besonders dann notwendig wenn drei farbige Sprites verwendet werden sollen. Große dithered HighRes Sprites können oft besser aussehen als 16 farbige LowRes Sprites.

Beachte dass es unrealistisch ist 4 Bitplanes und mehr als 3 Sprites zu verwenden. Das Ergebnis der Justierung die hierfür notwendig wäre, wäre ein wirklich schmales Display.

FetchMode

AGA Hardware gestattet es dem DMA Bitplane Daten in 16, 32 oder 64 Pixel-Gruppen zu verwalten. Je größer die Gruppen sind, umso mehr Bandbreite bleibt dem Prozessor. Besonders auffällig wird dies auf AGA Amigas ohne Fastram.

Die verwendeten Bitmaps müssen immer ein Vielfaches der eingesetzten Fetchmode sein. Der Versuch höhere Fetchmodes als 3 (64 Pixel) einzusetzen wird in erheblichen Schwierigkeiten enden mit `DisplayAdjust` noch ein brauchbares Display hinzubekommen. Ein Display mit Fetchmode 3 und mehr als einem Sprite dürfte auf eine Breite von 256 Pixel schrumpfen.

Mehrere Displays

Wenn mehr als eine Copperliste verwendet werden soll ist darauf zu achten dass zwischen den einzelnen Copperlisten eine Lücke von mindestens 3 Zeilen bestehen muss. Das heißt dass die Y-Position der unteren Copperliste der Höhe der vorherigen, plus der Y-Position der vorherigen, plus 3 betragen muss.

```
nextCopListY = lastCopListY + lastCopListHeight + 3
```

Erweiterte Copper-Kontrolle

Die erweiterte Version von `InitCopList` erlaubt die Zuweisung von benutzerdefinierten Copper-Befehlen. In der Display-Library befinden sich einige Funktionen die diesen Parameter benötigen.

Es gibt zwei Arten von Copper-Befehlen. Die erste erlaubt es dem Copper jede einzelne Zeile des Displays zu beeinflußen, während die zweite es dem Copper gestattet auf einer ganz bestimmten Zeile zu arbeiten.

Die folgenden Funktionen benötigen einen negativen Offset, dies zeigt die Anzahl der Instruktionen an die jeder Zeile zugewiesen werden muss.

```
DisplayDblScan CopList#,Mode[,copoffset]                    ;(size=-2)
DisplayRainbow CopList#,Register,Palette[,copoffset]        ;(ecs=-1 aga=-4)
DisplayRGB CopList#,Register,line,r,g,b[,copoffset]         ;(ecs=-1 aga=-4)
DisplayUser CopList#,Line,String[,CopOffset]                ;(size=-len/4)
DisplayScroll CopList#,&xpos.q(n),&xpos.q(n)[,CopOffset]    ;(size=-3)
```

Die folgenden Funktionen erwarten den Offset als positiven Wert welcher ausdrückt dass so viele Instruktionen für jede Instanz dieses Befehls zugeordnet werden. Beachte dass die obigen Funktionen nicht mit den jetzt folgenden vermischt werden dürfen:

```
CustomColors CopList#,CCOffset,YPos,Palette,startcol,numcols
CustomString CopList#,CCOffset,YPos,Copper$
```

Im Verzeichnis „Blitz Examples“ befinden sich Beispiele die diese Funktionen verwenden.

Beispiel 1

Das erste Beispiel erzeugt zwei große Bitmaps, wobei in eine Linien und in die andere Rechtecke gezeichnet werden. Aus der erzeugten 32 Farben Palette benutzt die erste Bitmap die ersten 16 und die zweite Bitmap die restlichen 16 Farben.

Die Flags für `InitCopList` summieren sich aus folgender Liste:

```
#eightbitplanes   = $08
#smoothscrolling  = $10
#dualplayfields   = $20
#lores            = $000
#fetchmode3       = $3000
#agacolors        = $10000
```

Beachte dass `InitCopList` vor dem Wechsel in den *BlitzMode* ausgeführt werden kann. Alle Display-Funktionen sind unabhängig vom aktiven Modus, mit Ausnahme von `CreateDisplay` welche im *BlitzModus* ausgeführt werden muss.

Auch interessant ist die Verwendung der erweiterten Form von `DisplayBitmap`. Sie gestattet es die Koordinaten für beide Bitmaps mit einem einzigen Befehl zuzuweisen.

```
; zwei 16 Farben Playfields im Dualplayfield-Modus

BitMap 0,640,512,4
BitMap 1,640,512,4
```

```
For i=0 To 100
      Use BitMap 0:Box Rnd(640),Rnd(512),Rnd(640),Rnd(512),Rnd(16)
      Use BitMap 1:Line Rnd(640),Rnd(512),Rnd(640),Rnd(512),Rnd(1)
Next

InitPalette 0,32
For i=1 To 31:AGAPalRGB 0,1,Rnd(256),Rnd(256),Rnd(256):Next

InitCopList 0,$13038

BLITZ

CreateDisplay 0
DisplayPalette 0,0

While Joyb(0)=0
      VWait
      x=160+Sin(r)*160:y=128+Cos(r)*128
      DisplayBitMap 0,0,x,y,1,320-x,256-y
      r+.05
Wend

End
```

Beispiel 2

Das zweite Beispiel demonstriert die Benutzung von Sprites. Die Funktion `DisplayAdjust` wird benötigt damit wir Zugriff auf alle 8 Sprite-Kanäle erhalten. Leider ist es schwierig eine höhere Fetchmode zu verwenden ohne auf ein sehr schmales Display zugreifen zu müssen.
`SpriteMode2` teilt Blitz mit dass wir 64 Pixel große Sprites für alle Sprite-Kanäle erzeugen wollen. Ohne SpriteMode würde sonst jeder Sprite 4 Kanäle beanspruchen. Eines der verbesserten Features von AGA.
Man beachte dass `DisplaySprite` auch Teil-Pixel Positionen als X-Parameter gestattet und dann versucht das Sprite an dieser Position darzustellen.

```
; sanft scrollender 16 Farben Screen mit 8x64 großen Sprites.

SpriteMode 2
InitShape 0,64,64,2:ShapesBitMap 0,0
Circlef 32,32,32,1:Circlef 16,8,6,2:Circlef 48,8,6,3:Circlef 32,32,8,0
GetaSprite 0,0

BitMap 0,640,512,4

For i=0 To 100
      Use BitMap 0:Box Rnd(640),Rnd(512),Rnd(640),Rnd(512),Rnd(16)
Next

InitPalette 0,48

For i=1 To 31:AGAPalRGB 0,i,Rnd(256),Rnd(256),Rnd(256):Next

InitCopList 0,$10014
DisplayAdjust 0,-2,8,0,16,0  ; underscan!

BLITZ
```

```
CreateDisplay 0
DisplayPalette 0,0

For i=0 To 7
      DisplaySprite 0,0,20+i*30,(20+i*50)&127,i
Next

While Joyb(0)=0
      VWait
      x=160+Sin(r)*160:y=128+Cos(r)*128
      DisplayBitMap 0,0,x,y
      r+.05
Wend

End
```

Befehlsreferenz

Programmfluss

Einführung

Ein Computer-Programm besteht aus Abfolgen von Befehlen die in ihrer Reihenfolge abgearbeitet werden. Einige Befehle unterbrechen diesen Fluß und lassen das Programm an eine andere Stelle springen um dort mit der Ausführung fortzusetzen. Es gibt zahlreiche Befehle um den Programmfluß in Blitz zu steuern.

Standard BASIC Befehle die den Programmfluß steuern (wie `Goto` oder `Gosub`) gibt es auch in Blitz. Der Unterschied ist dass sie anstelle von Zeilennummern sogenannte Labels als Ziel verwenden. Moderne BASIC Features wie Prozeduren (auch bekannt als Statements und Funktionen), `While...Wend`, `Repeat...Until`, `Select...Case` usw. erlauben eine strukturiertere Programmierung.

Zusätzlich gestattet es Blitz auf Interrupts zu reagieren, externe Ereignisse nach denen man zeitweise in andere Teile des Codes springen kann um dort dem Ereignis entsprechenden Code auszuführen.

Goto – Springe zu Programm-Label

Der Befehl `Goto` lässt das Programm zu einem bestimmten Label im Code zu springen und dort mit der Ausführung des Programms fortzufahren. Beachte das `Goto` keinen automatischen Rücksprung vorsieht. `Goto` „überspringt“ und/oder „wiederholt“ Teile des Codes.

Gosub – Springe zum Label und kehre zurück

Der Befehl `Gosub` arbeitet in zwei Schritten:

1. Die Anweisung im Source die auf `Gosub` folgt wird in einem speziellen Speicher, *Stack* genannt, gesichert.
2. Das Programm springt zum Label das bei `Gosub` angegeben wurde. Der Teil des Programms der mit `Gosub` angesteuert wird nennt sich „Unterroutine“ und wird mit der Anweisung `Return` beendet. `Return` sorgt dafür dass das Programm an die Stelle zurück springt die zuvor auf den Stack geschrieben wurde, also exakt die Stelle die auf das vorherige `Gosub` folgte.

```
Gosub myLabel

.myLabel
      ; bringe die Magie dieser Unterroutine zum Wirken
      Return
```

Return – I'll be back

`Return` wird verwendet um nach einem Sprung per `Gosub` aus der Unterroutine zurückzukehren und das Programm an der Stelle weiter arbeiten zu lassen die auf das `Gosub` folgte. Dies gestattet das Erstellen von

Unterroutinen die das Programm in logische Elemente aufteilt. Unterroutinen könnten in einem Spiel bspw. „Initialisierung“, „Menü“ und „GameLoop“ sein.

On Ausdruck Goto|Gosub Program Label[,Program Label...]

`On` lässt ein Programm verzweigen, abhängig vom Ergebnis des Ausdrucks. Ist das Ergebnis des Ausdrucks 1 wird das erste Label angesprungen, ist es 2 wird das zweite Label verwendet usw. Ist das Ergebnis des Ausdrucks kleiner als 1 oder es wurden nicht genug Labels übergeben, wird die Programmausführung ohne Sprung fortgesetzt.

```
On 4/2 Goto label1, label2

.label1
      ; der Ausdruck ergab 1

.label2
      ; der Ausdruck ergab 2
```

End – Programm beenden

Der Befehl `End` stoppt den gesammten Programmfluß unverzüglich. Wurde das Programm aus dem Blitz-Editor *TED* heraus gestartet kehrt das Programm dorthin zurück. Wurde das Programm als ausführbare Datei gestartet kehrt es zur Workbench oder in das CLI zurück, je nachdem von wo es gestartet wurde.

Stop – Programm anhalten

Der `Stop` Befehl bringt den Debugger dazu das Programm zu unterbrechen. Mit `Stop` hat man die Möglichkeit sogenannte Breakpoints in sein Programm zu setzen, also Stellen an denen die Programmausführung angehalten werden soll damit der Programmierer sich den Speicher anzusehen oder die Ausführung des Programms schrittweise fortzusetzen.
Damit Stop funktioniert muss in den Compiler Optionen „Runtime Errors“ eingeschaltet sein. Ein Klick auf den Button „Run“ im Debugger setzt die Programmausführung fort.

If Ausdruck [Then...] – Wenn Ausdruck = „Wahr“ dann....

Der Befehl `If` gestattet es Teile des Programms auszuführen, in Abhängigkeit vom Ergebnis des Ausdrucks. Der Befehl `Then` gibt an dass der ihm nachfolgende Teil des Codes nur ausgeführt wird wenn der Ausdruck „*Wahr*“ (also `True`) ergibt. Wird das `Then` weg gelassen, wird der gesamte nachfolgende Code (auch mehrere Zeilen) bis zum Auftreten eines `EndIf` ausgeführt.

```
x.l = 123
If x > 100 Then Print „X ist mindestens 100 groß“

If x > 100
      Print „X ist mindestens 100 groß“
EndIf
```

EndIf

Dieser Befehl wird verwendet um einen `If`-Block zu beenden. Ein `If`-Block der mit `EndIf` beendet wird kommt ohne ein `Then` aus und darf sich über mehrere Zeilen erstrecken. `If`-Blöcke dürfen weitere `If`-Blöcke beinhalten.

```
x.l = 123
If x > 100
      Print „X ist mindestens 100 groß“
EndIf
```

Else [Statement...]

Dieser Befehl kann verwendet werden um alternativen Code auszuführen falls der Ausdruck eines `If`-Blocks „Falsch“ (also `False`) ergibt.

```
x.l = 123
If x > 100
      Print „X ist größer als 100“
Else
      Print „X ist kleiner als 100“
EndIf
```

While Ausdruck – Solange es wahr ist

Dieser Befehl wird verwendet um einen Teil des Programm-Codes solange auszuführen wie der Ausdruck `True` ergibt. Der nachfolgende Programmteil wird mit einem `Wend` abgeschlossen. `Wend` lässt die Programmausführung zum einleitenden `While` zurück springen. Ist der Ausdruck `True` wird `Wend` übersprungen und die `While`-Schleife ist beendet.

```
x.l = 0
While x < 100
      x = x + 1
Wend

; wenn x = 100 ist macht das Programm hier weiter.
```

Wend

`Wend` wird in Verbindung mit `While` verwendet um einen Programmteil zu definieren der solange ausgeführt wird bis der Ausdruck für `While False` ergibt.

```
x.l = 0
While x < 100
      x = x + 1
Wend

; wenn x = 100 ist macht das Programm hier weiter.
```

Select Ausdruck – Wer ist der Richtige?

`Select` berechnet einen Ausdruck und merkt sich das Ergbnis. Innerhalb eines `Select`-Blocks werden mehrere `Case` definiert die das Ergebnis prüfen. Trifft das Ergbnis auf ein `Case` zu wird der nachfolgende Code ausgeführt.

```
x.l = 100
Select x
      Case 98
            Print „x ist 98“
      Case 99
            Print „x ist 99“
      Case 100
            Print „x ist 100“
End Select
```

Case Ausdruck

Ein Case folgt immer einem Select und prüft das durch Select Ausdruck erzeugte Ergebnis. Ist dieses Ergebnis gleich dem Wert den man hinter Case angibt so wird der nachfolgende Code ausgeführt.

```
x.l = 100
Select x
      Case 98
            Print „x ist 98“
      Case 99
            Print „x ist 99“
      Case 100
            Print „x ist 100“
End Select
```

Default

Platziert man ein `Default` in einem `Select`-Block so wird der Code nach `Default` ausgeführt wenn kein `Case` zuvor mit dem Ergebnis des Select übereinstimmte.

```
x.l = 101
Select x
      Case 98
            Print „x ist 98“
      Case 99
            Print „x ist 99“
      Case 100
            Print „x ist 100“
      Default
            Print „x ist irgend etwas anderes.“
End Select
```

End Select

Dieser Befehl beendet einen `Select`-Block und lässt das Programm mit dem nachfolgenden Code fortführen.

For Var=Ausdruck1 To Ausdruck2 [Step Ausdruck3]

Der `For` Befehl initialisiert eine `For...Next` Schleife. Alle `For...Next` Schleifen beginnen mit einem `For` und werden durch ein `Next` abgeschlossen. Der Code dazwischen wird solange ausgeführt wie *Ausdruck1* ungleich *Ausdruck2* ist. Wenn `For` ausgeführt wird nimmt „*Var*“ das Ergebnis von *Audruck1* an, man nennt dies auch die Index-Variable. Danach wird *Ausdruck2* ausgewertet und mit „*Var*“ verglichen. Sind die Ergebnisse nicht gleich wird der Inhalt der `For`-Schleife bis zum `Next` ausgeführt und die Index-Variable „*Var*“ um 1 erhöht. Solange bis „*Var*“ und *Ausdruck2* gleich sind.

Normalerweise erhöht sich „*Var*“ mit jedem Durchlauf der Schleife um 1. Mit dem Befehl `Step` kann dies aber auch geändert werden. Ein `Step` von 2 bewirkt dass nach jedem Schleifendurchlauf der Wert von „*Var*“ um 2 erhöht wird. Genausogut kann `Step` auch -1 sein und man durchläuft die Schleife „rückwärts“ weil „*Var*“ nun nicht mehr um 1 erhöht, sondern um 1 verringert wird. Beispielsweise kann man so eine Schleife realisieren die von 10 nach 0 zählt.

```
; schreibt 10x „Hello World“
For i = 0 To 10
      Print „Hello World
Next

; schreibt 5x „Hello World“
For i = 0 To 10 Step 2
      Print „Hello World“
Next

; schreibt auch 10x „Hello World“ … aber heimlich rückwärts
For i = 10 To 0 Step -1
      Print „Hello World“
Next
```

Next

Der Befehl `Next` beendet eine `For...To` Schleife.

Repeat

Dieser Befehl wiederholt, wie die `While`-Schleife, einen Code-Block bis ein bestimmter Ausdruck `True` (*wahr*) ergibt. `Repeat` leitet eine `Repeat...Until` Schleife ein. Im Gegensatz zu einer `While...Wend` Schleife wird der Code zwischen `Repeat` und `Until` auf jedenfall mindestens einmal ausgeführt. Das liegt daran dass erst das abschliessende `Until` prüft ob die Bedingung zum Abbruch der Schleife wahr (`True`) oder falsch (`False`) ist.

```
x = 0
Repeat
      x = x + 1
      Print „Hello World"
Until x = 10      ; bis (until) x = 10 ist, springe zurück zu Repeat
```

Until Ausdruck – Bis es wahr wird.

`Until` wird verwendet um eine `Repeat...Until` Schleife abzuschliessen. Die Schleife wird solange ausgeführt bis der Ausdruck (bspw „x = 10“) `True` (*wahr*) ergibt. Sobald der Ausdruck *wahr* ist wird das Programm nach `Until` fortgesetzt.

Forever - Endlosschleife

Dieser Befehl kann als Ersatz für `Until` in einer `Repeat...Until` Schleife eingesetzt werden. Es zwingt die Schleife dazu für immer (forever) ausgeführt zu werden. `Forever` entspricht einem `Until 0`.

Pop Gosub|For|Select|If|While|Repeat

Manchmal kann es notwendig sein eine Schleife vorzeitig zu beenden, obwohl der bedingende Ausdruck noch nicht `True` ist. So beendet Pop For beispielsweise eine For...To Schleife sofort und springt in den nachfolgenden Code.

```
x = 0
For i = 0 To 10
      x = x + 1
      If x = 5 Then Pop For
Next

; hier macht das Programm dann weiter.
```

MouseWait – Lauscht der Maus

Dieser Befehl stoppt die Ausführung des Programms solange bis der Benutzer die linke Maustaste drückt. Ist die linke Maustaste, in dem Moment in dem das Programm ein `MouseWait` erreicht, bereits gedrückt wird das `MouseWait` ignoriert und das Programm fortgesetzt.
`MouseWait` sollte nur zum testen des Programms eingesetzt werden da es das Multitasking eines Amiga recht stark verlangsamt.

VWait [Frames]

`VWait` hält das Programm solange an bis der Kathodenstrahl (Video-Strahl) wieder die erste Zeile des Bildschirms erreicht. Der optionale Parameter *Frames* erlaubt es eine Anzahl von Durchläufen anzugeben

die der Strahl unternehmen soll bevor das Programm fortgesetzt wird. Beispiel: Ein PAL Amiga baut den Bildschirm 50 Mal pro Sekunde neu auf. Ein `VWait 50` würde das Programm also exakt eine Sekunde lang warten lassen.

`VWait` ist sehr nützlich um Animationen mit dem Bildschirm zu synchronisieren. So kann man sicher stellen dass keine Änderungen am Inhalt eines Displays vorgenommen werden solange das Bild noch aufgebaut wird.

Statement Name{ [Parameter 1[, Parameter 2...]] }

`Gosub` und `Goto` wurden bereits besprochen, `Statement` ist die nächste Stufe dieser Unterroutinen. Während man mit `Gosub` und `Goto` schlicht an eine andere Stelle des Programms springt, kann man einem `Statement` noch Parameter übergeben mit denen dieses `Statement` arbeiten soll. Sicher könnte man auch globale Variablen einsetzen und eine Unterroutine damit arbeiten lassen. Aber: Ein `Statement` kann man in ein neues Programm mitnehmen!

Ein `Statement` hat einen Namen, kann bis zu 6 Parameter empfangen und muss definiert sein bevor man es aufruft. Das bedeutet: Der Aufruf eines Statements kann im Quelltext nicht über dem `Statement` stehen sondern immer nur darunter.

```
Greetings{„Hello World"}          ; Fehler: Statement nicht definiert

Statement Greetings{ greets$ }
      Print greets$
End Statement

Greetings{„Hello World"}          ; so funktionierts
```

End Statement

Ein `End Statement` schliesst ein `Statement` ab welches durch `Statement{[optional bis zu 6 Parameter]}` eingeführt wurde. Alle Statements müssen durch ein `End Statement` abgeschlossen werden.

Statement Return

Dieser Befehl wird dazu benutzt um die Ausführung eines Statements vorzeitig zu beenden. Beachte dass ein `Statement` keinen Rückgabewert besitzt und auch `Statement Return` keinen Wert zurück liefern kann. Wird die Ausführung eines Statements beendet (vorzeitig oder nicht), wird das Programm an der Stelle fortgesetzt die nach dem Aufruf des Statements folgt.

Function [.Type] Name{ Parameter1[, Parameter2...]] }

Funktionen ähneln Statements, nur dass sie auch einen Wert zurückgeben können. Desweiteren werden sie mit einem `End Function` abgeschlossen. Der optionale `.Type` gibt an von welchem Datentyp der Rückgabewert ist. Es ist zu beachten dass eine Funktion einen Wert zurück geben *kann* aber nicht *muss*.

Wird ein Wert zurückgegeben ohne dass `.Type` angegeben wurde, wird der Wert als Standard-Datentyp zurück gegeben. Greift man selbst nicht ein ist dies *Quick* (.q).
Wie Statements darf eine Funktion bis zu sechs Parameter besitzen. Diese dürfen nur den sechs Basis-Datentypen entsprechen. Eigene Datentypen (`NewType`) dürfen nicht ohne Weiteres als Parameter verwandt werden.
Das Ergebnis einer Funktion wird mit dem Befehl `Function Return` zurück gegeben.
Auch Funktionsaufrufe dürfen erst erfolgen wenn die Funktion definiert ist.

```
x.l = Add{2, 3}              ; Fehler: Add{} ist noch nicht bekannt

Function.l Add{ first.l, second.l}
      result.l = first + second
      Function Return result
End Function

x.l = Add{2, 3}              ; hier funktionierts, x ist nun 5
```

End Function

Eine Function wird mit `End Function` beendet. Jede Funktion muss mit mit einem `End Function` geschlossen werden.
Function Return Ausdruck
Function Return erlaubt es einer Funktion einen Wert an den Aufrufer zurückzugeben. So kann die Funktion bspw. einen oder mehrere Parameter entgegen nehmen, aus diesen einen neuen Wert berechnen und diesen Wert zurück geben so dass der Aufrufer mit ihm weiter arbeiten kann.

```
Function.l Add{ first.l, second.l}
      result.l = first + second
      Function Return result
End Function

x.l = Add{2, 3}
```

x.li

Shared Var[, Var...]

Normalerweise kennen Statements die Namen von Variablen nicht die sich nicht innerhalb der Statement Definition befinden. So kann das Hauptprogramm eine Variable namens „*einWert*“ kennen und das Statement darf ebenfalls eine Variable namens „*einWert*“ definieren ohne dass die Hauptprogramm-Variable „einWert“ mit der eigenen Definition kollidiert.
`Shared` erlaubt es nun aber dem Statement eine globale Variable in der eigenen Definition bekannt zu machen und damit zu arbeiten. Dies wird bspw. dann genutzt wenn man mehr als nur 6 Parameter für ein `Statement` benötigt.

```
x.l = 100
Statement IncrementX{value.l}
      Shared x.l
      x = x + value
End Statement

IncrementX{100}              ; x ist nun 200
```

Setint Type – Interrupts zu mir

`Setint` wird dazu benutzt um auf Ereignisse (Interrupts) zu reagieren die außerhalb des eigenen Programms passieren. Es kommt oft vor dass ein Programm läuft und plötzlich passiert in den Weiten des Computers etwas auf das man gern eine Antwort geben würde. Also definiert man einen `Setint`-Block, welcher mit `End Setint` abgeschlossen wird und innerhalb dieses Blocks wird dann der Code ausgeführt der auf das entsprechende Ereignis reagieren soll.

Der Parameter `Type` von `Setint` kann einen der folgenden Werte annehmen:

0 serieller Übertragungspuffer ist leer
1 Disk-Block gelesen oder geschrieben
2 Software Unterbrechung (Software Interrupt)
3 CIA Ports Interrupt
4 Koprozessor („Copper“) Interupt
5 Vertical Blank Interrupt (der Bildschirm wurde neu gezeichnet)
6 der Blitter ist mit seiner Arbeit fertig
7 Audio Kanal 0 Pointer / Länge geholt
8 Audio Kanal 1 Pointer / Länge geholt
9 Audio Kanal 2 Pointer / Länge geholt
10 Audio Kanal 3 Pointer / Länge geholt
11 serieller Empfangspuffer ist voll
12 Diskettenlaufwerk ist synchronisiert
13 Unterbrechung (Interrupt) von aussen

Der nützlichste all dieser Interrupts ist der *VBlank* Interrupt. Dieses Ereignis wird immer dann ausgelöst wenn der Bildschirminhalt neu gezeichnet wurde. Auf PAL Systemen also 50 mal pro Sekunde.

Interrupt-Handler (also die `Setint` Blöcke die diese Ereignisse behandeln) müssen so schnell wie möglich ausgeführt werden da es sein kann dass der nächste Interrupt bereits stattfindet während das eigene Programm sich noch durch den Handler arbeitet.

Interrupt-Handler dürfen niemals auf String-Variablen oder Literale zugreifen. Für den *BlitzModus* ist dies die einzige Einschränkung. Für den *AmigaModus* kommen noch Blitter-Operationen, Dateizugriffe und Intuition-Funktionen dazu.

Um einen Interrupt-Handler zu implementieren muss ein `Setint`-Block aufgesetzt werden der den entsprechenden Code zur Verarbeitung des Interrupts beinhaltet. Ein `End SetInt` beendet den Handler. Dem einführenden `SetInt` folgt einer der Codes aus obiger Tabelle. Als Beispiel reagiert ein `SetInt 5` auf den Interrupt *VBlank-Interrupt*. Es können auch mehrere Handler auf denselben Interrupt reagieren.

End Setint

Dieser Befehl beendet einen Interrupt-Handler welcher mit `Setint` initialisiert wurde. Auf jedes `Setint` muss auch ein `End Setint` folgen damit Blitz weiß wo der Handler zuende ist.

ClrInt Type

Dieser Befehl wird verwendet um alle Interrupt-Handler für den Typ *Type* zu entfernen. Wurden zuvor mittels Setint Interrupt-Handler für einen bestimmten Interrupt definiert, so können diese mit `ClrInt` wieder gelöst werden. Dies ist nützlich wenn man nur für einen bestimmten Zeitraum auf Interrupts reagieren möchte und danach die Rechenzeit für andere Unterroutinen benötigt oder wenn die Interrupts dann schlicht nicht mehr

von Belang sind.

SetErr – Fehler zu mir

Mit `SetErr` hat man die Möglichkeit eigene Handler für Laufzeitfehler in Blitz zu schreiben. Zwar behandelt Blitz die Laufzeitfehler immer noch selbst, aber danach kommen die eigenen Handler zum Einsatz. Ein `SetErr` muss immer mit einem `End SetErr` abgeschlossen werden.

End SetErr

Dieser Befehl muss immer als Abschluss eines `SetErr`-Blocks stehen, ansonsten würde auch nachfolgender (wahrscheinlich nicht dazugehöriger) Code als Fehler-Handler angesehen. In der Regel gibt es für jeden Block-Typ auch ein `End „Funktionsname“`.

ClsErr

Mit `ClsErr` wird ein Error-Handler entfernt der zuvor mittels `SetErr` definiert wurde.

ErrFail

Diese Funktion wird verwendet um in Error-Handlern einen „normalen“ Fehler auszulösen der bewirkt dass der Code nach dem Error-Handler weiter ausgeführt wird. (← hier gibt es ein Übersetzungsproblem, ich bin dankbar für jede Erklärung)

Anhang 1

Der Blitz2 Editor TED

- Texteingabe
- Textblöcke markieren
- Die Editor-Menüs
- Der Blitz2 File-Requester
- Tastaturkürzel
- Das Compiler-Menü
- Die Compiler-Optionen

Einleitung

Um Programmcode eingeben und compilieren zu können, braucht man einen sogenannten Editor. Der Blitz2-Editor *Ted* dient sowohl als Schnittstelle zum Compiler als auch als eigener Editor für ASCII-Texte (ASCII ist der Standard für normalen Text).

Im folgenden wird *Ted* in seiner Funktion als eigener ASCII-Editor beschrieben. Um *Ted* zu starten klicken Sie entweder auf das Ted-Symbol oder geben Sie `ted` in der Kommandozeile ein.

Wenn der Editor gestartet ist, sollte folgendes Bild auf dem Schirm erscheinen:

>>> screen shot editor screen <<<

Die senkrechten und waagerechten Balken werden „Scrollbars“ (Rollbalken) genannt. Wenn der Text länger oder breiter als das Fenster ist, kann der Rest des Textes in das Fenster „gescrollt“ werden. Dies geschieht, indem man die Scrollbars mit dem linken Mausknopf verschiebt.

Unten im Bildschirm befindet sich eine Statuszeile, die Informationen über die Cursor-Position (bezogen auf den Dateianfang) und die Größe des verfügbaren Speicher des Amiga liefert.

Mit dem linken Mausknopf kann das Editor-Fenster, wie jedes andere Amiga-Fenster auch, vergrößert werden und mit dem Vordergrund/Hintergrund-Knopf (rechts oben) kann das Fenster nach vorne geholt oder in den Hintergrund geschoben werden.

Texteingabe

Die Texteingabe im Editor erfolgt wie bei einer normalen Schreibmaschine, man tippt einfach drauflos und mit der „Return“-Taste wird in die nächste Zeile gesprungen.

Dabei bewegt sich der Cursor (Textmarke), das kleine Kästchen auf dem Bildschirm, mit jedem eingegebenen Zeichen weiter. Der Cursor bestimmt die Stelle, an der der Text, den Sie tippen, eingefügt wird.

Mit Hilfe der Pfeiltasten kann der Cursor innerhalb des Textes umherbewegt werden.

Alle Eingaben werden dort in den Text eingefügt wo sich der Cursor gerade befindet. Dabei wird der Text, der sich rechts vom Cursor befindet entsprechend nach rechts und unten verschoben.

Ebenso kann das Zeichen, auf dem sich er Cursor steht mit der *DEL*-Taste gelöscht werden und der nachfolgende Text füllt den leeren Raum wieder auf.

Die Taste links von der *DEL*-Taste dient ebenfalls zum Löschen, hierbei wird aber das Zeichen links vom Cursor gelöscht.

Die Tabulator-Taste *TAB* funktioniert ähnlich wie bei einer Schreibmaschine, der Cursor und der dahinter befindliche Text werden um eine bestimmte Anzahl Spalten nach rechts geschoben.

Mit der *RETURN*-Taste wird eine neue Zeile begonnen. Soll der Text rechts von der Cursor-Position in eine neue Zeile geschoben werden, muß *SHIFT-RETURN* gedrückt werden. Dadurch wird ein Zeilenvorschub in die Zeile eingefügt.

Wird die *SHIFT*-Taste zusammen mit einer der Pfeiltasten gedrückt, bewegt sich der Cursor an den Anfang oder das Ende der Zeile (links, rechst) bzw. an den Anfang oder das Ende des Bildschirms (oben, unten).

In Anhang 2 befindet sich eine Liste der Tastaturkürzel, die die Bedienung erleichtern.

Textblöcke markieren

Beim Programmieren ist es oft notwendig ganze Textblöcke zu bearbeiten. Hierzu muß der Textblock zunächst markiert werden. Dies kann entweder mit der Maus oder mit einer Funktionstaste geschehen.

Bewegen Sie den Mauszeiger an den Anfang des Textblocks, drücken Sie die linke Maustaste und bewegen Sie die Maus mit gedrückter Taste an das Ende des Blocks. Erst dann lassen Sie den Mausknopf los.

Alternativ dazu können Sie den Anfang des Textblocks mit der Taste *F1* markieren und das Ende mit *F2*.

Eine nützliche Einrichtung für das Programmieren ist die Tastenkombination *Amiga-A*, mit der die aktuelle Zeile und alle anschließenden Zeilen, die genausoweit eingerückt sind, markiert werden.

Die Editor-Menüs

Mit dem rechten Mausknopf wird die Menüleiste des Blitz2 Editors aufgerufen. Es folgt eine Beschreibung der einzelnen Funktionen, die über die Menüs erreichbar sind (von links nach rechts):

Das PROJECT-Menü

- **NEW** löscht den gerade im Editor befindlichen Text aus dem Speicher. Wenn sich der Inhalt seit der letzten Sicherung geändert hat, erscheint ein Requester, der eine Bestätigung für das Löschen verlangt.

- **LOAD** lädt eine Datei von der Platte. Es erschient ein File-Requester, der eine einfache Auswahl der Datei ermöglicht. Am Ende dieses Kapitels befindet sich eine genauere Beschreibung des File-Requesters.

- **SAVE** speichert die im Editor befindliche Datei auf der Platte ab. Es erscheint ein File-Requester für die Auswahl des Dateinamens.

- **DEFAULTS** bestimmt das Aussehen des Blitz2 Editors. Hier wird die Farbauswahl und die Schriftgröße eingestellt und bestimmt, ob der Editor beim Speichern von Dateien ein Symbol (Icon) anlegen soll. Außerdem kann eingestellt werden, wie nahe der Cursor an den Rand des Bildschirms gelangen darf, bevor der Text gescrollt wird. Alle Einstellungen werden in der Datei

„1:BlitzEditor.opts“ gespeichert.

- **ABOUT** zeigt die Versionsnummer und Copyright-Vermerke.

- **PRINT** druckt die Datei auf dem Standard-Druckerkanal PRT:.

- **CLI** startet eine Kommandozeile (Command Line Interface) in der Sie Amiga-Befehle eingeben können. Mit dem Befehl `ENDCLI` kehren Sie zurück in den Editor.
- **CLOSEWB** schließt die Workbench, wenn diese geöffnet ist. Die ist nützlich, wenn der Speicherplatz sehr knapp wird, da hierdurch etwa 40KB frei werden.

- **QUIT** beendet *Ted* und Sie kehren zur Workbench oder zur Kommandozeile zurück.

Das EDIT-Menü

- **COPY** kopiert einen zuvor mit der Maus oder der *F1-F2*-Kombination markierten Textblock an die aktuelle Cursor-Position. Hierfür kann auch die Taste *F4* verwendet werden.

- **KILL** löscht den markierten Textblock, auch durch *SHIFT-F3* zu erreichen.

- **BLOCK TO DISK** speichert den markierten Textblock als ASCII-Datei auf der Platte ab.

- **INSERT FROM DISK** lädt eine Datei von der Platte und fügt sie an der aktuellen Cursor-Position in den Text ein.

- **FORGET** entfernt die Markierung eines Textblocks.

- **INSERTLINE** beginnt eine neue Zeile an der aktuellen Cursor-Position.

- **DELETE LINE** löscht die Zeile, auf der sich der Cursor gerade befindet.

- **DELETE RIGHT** löscht alles rechts vom Cursor bis zum Zeilenende.

- **JOIN** fügt die nächste Zeile an das Ende der aktuellen Zeile an.

- **BLOCK TAB** verschiebt den gesamten markierten Textblock um eine Tabulator-Position nach rechts.

- **BLOCK UNTAB** verschiebt den gesamten markierten Textblock um eine Tabulator-Position nach links.

Das SOURCE-Menü

- **TOP** positioniert den Cursor an den Anfang der Datei.

- **BOTTOM** positioniert den Cursor an das Ende der Datei.

- **GOTO LINE** positioniert den Cursor an die angegebene Zeile.

Das SEARCH-Menü

Der Blitz2 Editor kann eine Datei nach einer bestimmten Zeichenfolge, wie z.B. „HALLO“ absuchen.

- **FIND** sucht die Datei nach einer Zeichenfolge ab. Es erscheint der unten näher beschriebene Find-Requester.

- **NEXT** sucht nach dem nächsten Auftreten der zuvor mit FIND angegebenen Zeichenfolge.

- **PREVIOUS** sucht rückwärts nach dem vorherigen Auftreten der zuvor mit FIND angegebenen Zeichenfolge.

- **REPLACE** sucht eine Zeichenfolge und ersetzt sie durch eine andere. Es erscheint der selbe Requester wie bei FIND (s.u.).

- Bei der Auswahl von **FIND** erscheint der folgende Requester:

  >>> screen shot find requester <<<

  In das mit *FIND* bezeichnete Eingabefeld wird der gesuchte Text eingetragen, anschließend klicken Sie auf *NEXT*. Hiermit wird der Cursor an die Stelle positioniert, an der der Text gefunden wurde. Konnte der Text nicht gefunden werden, blinkt der Bildschirm auf.

  Mit dem *PREVIOUS*-Knopf können Sie von der aktuellen Position aus rückwärts in der Datei nach dem Text suchen

  Wenn Sie *CASE SENSITIVE* anwählen, wird die Groß- und Kleinschreibung in der gesuchten Zeichenfolge berücksichtigt. Voreingestellt ist keine Berücksichtigung der Großschreibung.

  Um eine Zeichenfolge durch eine andere ersetzen zu lassen, tragen sie den neuen Text in das Feld *REPLACE* ein. Wird der gesuchte Text anschließend mit *NEXT* oder *PREVIOUS* gefunden, wird er automatisch durch den neuen Text ersetzt.

  Mit *REPLACE ALL* wird die gesamte Datei nach dem Find-Text durchsucht und dieser automatisch jedes mal ersetzt.

Der Blitz2 File-Requester

Ein Requester ist eine Maske, in der der Benutzer aufgefordert wird, etwas einzugeben (die deutsche Übersetzung hierfür ist „Eingabeaufforderung“, wir lassen es deshalb lieber bei dem Begriff Requester). Ein File-Requester ist ein Standard-Dialog, der zur Eingabe oder Auswahl eines Dateinamens dient. Blitz2 verwendet einen eigenen File-Requester, der immer dann aufgerufen wird, wenn eine Datei von der Platte geladen oder auf der Platte abgespeichert werden soll.

Ein File-Requester wird beendet, indem auf das kleine Symbol links oben in der Ecke oder auf den *CANCEL*-Knopf rechts unten geklickt wird.

Die Liste zeigt ihnen alle in dem aktuellen Verzeichnis verfügbaren Dateien. Sie können eine Datei aus dieser Liste mit einem Doppelklick auf den entsprechenden Eintrag auswählen. Mit dem Schiebebalken (Scrollbars) rechts können sie durch die Liste rollen.

Wenn Sie auf einen mit <DIR> bezeichneten Eintrag klicken, wird in dieses Verzeichnis gewechselt und die Liste neu aufgebaut.

Der *PARENT*-Knopf dient dazu, in das übergeordnete Verzeichnis („Elternverzeichnis“) zu wechseln.

Mit dem *DRIVES*-Knopf wird eine Liste aller verfügbaren physikalischen und logischen Laufwerke (Volumes) oben in der Liste eingefügt, sodaß Sie auch diese auswählen können.

Sie können auch direkt einen Pfad- und Dateinamen in die entsprechenden Eingabefelder eintragen. Hierzu müssen Sie zunächst mit der Maus in das Feld klicken, um es zu aktivieren, bevor Sie den Namen eintippen können.

Wenn Sie Ihre Auswahl getroffen haben, klicken auf den *OK*-Knopf.

Eine Besonderheit des Blitz2 File-Requesters ist der *CD*-Knopf. Wenn Sie eine Datei laden, die sich in einem anderen Verzeichnis als dem aktuellen Verzeichnis des Editors befindet, so ändert sich letzteres normalerweise nicht. Manchmal kann es aber für weitere Dateizugriffe nützlich sein, daß auch das aktuelle Verzeichnis des Editors dorthin gewechselt wird. Hierzu dient der *CD*-Knopf.

Eine weitere ungewöhnliche Eigenschaft des Blitz2 File-Requesters ist die Möglichkeit, seine Größe zu verändern, indem die rechte untere Ecke des Fensters mit der Maus "gezogen" wird (den linken Mausknopf gedrückt halten und die Maus bewegen). Hierdurch vergrößert sich auch die Datei-Liste, was hilfreich sein kann wenn sich viele Dateien in einem Verzeichnis befinden.Das COMPILER-Menü
Der Blitz2 Editor dient nicht nur zur Texteingabe, sondern auch als Entwicklungsumgebung beim Programmieren. Hierzu können sämtliche Befehle, die zum Compilieren und Austesten eines Programms notwendig sind, direkt vom Editor aufgerufen werden, ohne diesen verlassen zu müssen.

COMPILE & RUN compiliert (übersetzt) das im Editor befindliche Programm direkt in den Speicher und startet es, wenn es fehlerfrei übersetzt werden konnte.

RUN startet ein bereits in den Speicher übersetztes Programm.

CREATE FILE compiliert das Programm und speichert es als ausführbare Datei auf der Platte ab.

OPTIONS wird im nächsten Abschnitt ausführlich beschrieben.

CREATE RESIDENT wandelt das im Editor befindliche Programm in eine „residente Datei“ um. Eine residente Datei enthält alle Makros, Konstanten und NewType-Definitionen eines Programms in pre-compilierter Form, sodaß diese nicht mehr im eigentlichen Programmcode erscheinen müssen. Dadurch erhöht sich die Geschwindigkeit, mit der compiliert wird.

VIEW TYPE dient dazu, die Typ-Definition einer Variablen anzusehen. Unter-Typen können ebenfalls angesehen werden.

CLI ARGUMENT dient dazu, Aufruf-Parameter an ein Programm zu übergeben, wenn dieses vom Editor aus gestartet wird, so als ob das Programm von einer Kommandozeile (CLI) aus aufgerufen wird.

CALCULATOR ist ein kleiner Taschenrechner, mit dem Sie Formeln (auch mit verschachtelten Klammern) ausrechnen können. Sie können im Dual- Dezimal- oder Hexadezimal-System rechnen. Zahlen, die zur Basis 2 genommen werden sollen, kennzeichnen Sie durch ein vorangestelltes %-Zeichen, Hexadezimal-Zahlen durch ein $-Zeichen.

RELOAD ALL LIBS lädt alle im Laufwerk BLITZLIBS: befindlichen Dateien erneut in den Speicher. Dies ist notwendig, wenn Sie ihre eigenen Libraries schreiben und diese testen wollen. Sie müssten sonst Blitz2 erneut starten.

Die Compiler-Optionen

Wenn Sie denn Menüpunkt *OPTIONS* anklicken, erscheint folgende Maske, die nachfolgend erläutert wird:

>>> screen shot options menu <<<

Create Icons for Executable Files
Wenn ein Programm mit *CREATE FILE* in ausführbarer Form auf der Platte gespeichert wird, kann gleichzeitig dafür ein Symbol (Icon) erzeugt werden. Nur wenn ein Symbol für das Programm existiert, kann das Programm von der WorkBench aus aufgerufen werden. Beachten Sie, daß hierfür der Befehl `WBStartUp` am Anfang des Programmcodes stehen muß.

Runtime Error Debugger

Dies schaltet die Überprüfung von Laufzeitfehlern ein. Tritt ein Laufzeitfehler auf, wird automatisch der Blitz2 Debugger aufgerufen. In Kapitel 5 befindet sich eine ausführliche Beschreibung der Laufzeitfehler.

Make Smallest Code
Der Blitz2 Compiler kann Programme dahingehend optimieren daß sie eine minimale Größe besitzen (zweiphasiges Compilieren). Wenn das Programm als ausführbare Datei auf der Platte gespeichert werden soll, kann die Optimierung mit dieser Option eingestellt werden. Allerdings dauert dann das Compilieren selbst länger.

Debug Info
Diese Option erzeugt eine Symboltabelle wenn das ausführbare Programm auf Platte gespeichert wird. Diese kann von Debuggern wie dem MetaScope von Metadigm verwendet werden.

Buffer Sizes
Wenn Blitz2 als einphasiger Compiler benutzt wird, können hier verschiedene Puffergrößen eingestellt werden. Bei zweiphasigem Compilieren mit der *Make Smallest* Option werden die Puffergrößen automatisch optimiert.

Eine Ausnahme bilden die String-Puffer. Wenn sehr große Strings verwendet werden (um z.B. eine gesamte Datei in einen String einzulesen), muß der Workspace-Puffer auf die Länge des größten verwendeten String eingestellt werden.

Object Maximums
Hiermit wird die maximale Anzahl der von Blitz2 verwalteten Objekte (Screens, Shapes, etc.) eingestellt. Siehe auch Kapitel 6.

Resident
Diese Option fügt die vor-compilierten residenten Dateien in die Blitz2 Umgebung ein. Es muß der Name der Datei eingegeben werden.

Anhang 2

Tastaturkürzel

Manchmal kann es lästig sein, nach der Maus greifen zu müssen, um einen bestimmten Befehl auszuführen. Daher existieren für viele Editor-Befehle auch Tastenkombinationen, die das gleiche bewirken.

Die Kombinationen verwenden alle die rechte „Amigataste“ (gleich rechts neben der Leerzeichen-Taste), die gleichzeitig mit der angegebenen Befehlstaste gedrückt werden muß.

Shortcut	**Befehl**	**Wirkung**
RAmiga + A	SELECT	markiert alle Zeilen vor und hinter der aktuellen Zeile, die genauso weit eingerückt sind, wie die aktuelle Zeile (dient zum schnelleren Markieren von Programmabschnitten).
RAmiga + B	BOTTOM	positioniert den Cursor auf die letzte Zeile der Datei.
RAmiga + D	DELETE LINE	löscht die Zeile, auf der sich der Cursor gerade befindet.
RAmiga + F	FIND / REPLACE	ruft das FIND-Kommando im SEARCH-Menü auf.
RAmiga + G	GOTO LINE	bewegt den Cursor zu der angegebenen Zeile.
RAmiga + I	INSERT LINE	fügt an der aktuellen Cursorposition einen Zeilenvorschub ein. Der Text rechts und unterhalb des Cursors kommt in eine neue Zeile.
RAmiga + J	JOIN LINE	fügt die nächste Zeile an das Ende der aktuellen Zeile an.
RAmiga + L	LOAD	liest eine Datei von der Platte und lädt sie in den Editor.
RAmiga + N	NEXT	sucht nach dem nächsten Auftreten des gesuchten Textes.
RAmiga + P	PREVIOUS	sucht rückwärts nach dem vorherigen Auftreten des gesuchten Textes.
RAmiga + Q	QUIT	beendet den Blitz2 Editor.
RAmiga + R	REPLACE	ersetzt den gefundenen Text durch den im Find-Befehl angegebenen Ersatz-Text.
RAmiga + S	SAVE	speichert die im Editor befindliche Datei auf der Platte ab.

RAmiga + T	TOP	positioniert den Cursor in die erste Zeile der Datei.
RAmiga + W	FORGET	hebt die Markierung eines Textblocks auf.
RAmiga + Y	DELETE TO RIGHT	löscht alle Zeichen rechts vom Cursor in der aktuellen Zeile.
RAmiga + Z	CLI	eröffnet eine Kommandozeile.
RAmiga + ?	DEFAULTS	erlaubt Änderungen der Voreinstellungen für das Aussehen des Blitz2 Editors.
RAmiga +]	BLOCK TAB	verschiebt den markierten Textblock um einen Tabulatorsprung nach rechts.
RAmiga + [	BLOCK UNTAB	verschiebt den markierten Textblock um einen Tabulatorsprung nach links.

Anhang 3

Programmiertechniken

- Namensgebung
- Anmerkungen und Kommentare
- Techniken der strukturierten Programmierung
- Modularisierung
- Nebenbei...
- Lesbarkeit des Programms

Dieses Kapitel enthält eine Reihe von Hinweisen und Tipps, die dazu dienen, richtig lauffähige Blitz2-Programme zu schreiben.

Namensgebung

Die Regeln
Bei der Vergabe von Variablennamen und Namen für Labels (Sprungmarken) müssen folgende Regeln beachtet werden:

- Namen können beliebig lang sein
- Namen müssen mit einem Buchstaben (A..Z, a..z) oder einem „Underscore“ (Unterstreichung) beginnen
- Namen dürfen nur alphanumerische Zeichen (Buchstaben und Zahlen) und Underscores enthalten
- Namen dürfen nicht mit den selben Buchstaben beginnen, wie einer der Blitz2-Befehle

Im übrigen wird bei Variablen- und Labelnamen die Groß- und Kleinschreibung berücksichtigt, d.h. die Variablen `meinschiff` und `MeinSchiff` haben nichts miteinander zu tun.

Stil
Es gibt viele Methoden der Namensgebung, die das Programmieren erleichtern. Im folgenden werden einige Richtlinien genannt, die dazu dienen, den Überblick zu behalten, wenn das Programm immer weiter anwächst und mehr und mehr Variablen und Labels verwendet werden.

Das wichtigste ist jedoch, den einmal gewählten Stil beizubehalten.

Es ist sinnvoll, Variablen und Labels in Gruppen zusammenzufassen und diese Gruppenzugehörigkeit in die Namensgebung einfließen zu lassen. Dadurch erhöht sich der Informationsgehalt der Namen. Gruppen können folgendermaßen gekennzeichnet werden:

- nur Großbuchstaben „NAME“, große Anfangsbuchstaben „Name“ oder nur Kleinbuchstaben „name“
- einzelne Buchstaben „l“, Worte „Loop“ oder Doppelworte „MainLoop“
- Underscore am Anfang „_loop“ oder in der Mitte „main_loop“
- numerische Zusätze wie „loop1“, „loop2“ usw.

Die Namensgebung ist eine Frage des persönlichen Stils, aber durch die Beibehaltung eines Stils und die Verwendung von „sinnvollen“ Namen kann die Lesbarkeit eines Programms wesentlich erhöht und die Fehlersuche erleichtert werden.

Häufige Probleme bei der Namensgebung
Der folgende Abschnitt beschreibt einige der Probleme, die bei schlechter Namensgebung auftreten können.

Wenn irrtümlich auf eine noch nicht vereinbarte Variable zugegriffen wird (weil der falsche Name verwendet wurde), meldet der Compiler keinen Fehler, sondern legt eine neue Variable an und initialisiert sie mit dem Wert 0. Diese Gefahr kann durch eine einheitliche Namenskonvention verringert werden.

Es verlangsamt die Programmentwicklung, wenn immer wieder zum Anfang des Programmcode zurückgegangen werden muß, weil vergessen wurde, wie die Variablen hießen. Eine handgeschriebene Liste neben der Tastatur erleichtert das Nachschlagen.

Die Verwendung von langen Namen erhöht zwar die Lesbarkeit, birgt aber auch die Gefahr von vermehrten Tippfehlern.

Die Vergabe von vulgären oder obszönen Namen kann zwar ganz lustig sein, sollte aber vermieden werden, wenn andere Leute den Code lesen.

Anmerkungen und Kommentare

Im Gegensatz zu anderen BASIC-Versionen wird bei Blitz2 nicht die Anweisung *REM* verwendet, um einen Kommentar zu kennzeichnen, sondern das Semikolon. Alles was hinter einem Semikolon bis zum Ende der Zeile folgt, wird als Kommentar betrachtet und vom Compiler ignoriert. Auf diese Weise können Programme dokumentiert werden.

Wenn alle Routinen entsprechend kommentiert sind, können diese in zukünftigen Projekten weiterverwendet werden. Nichts ist so lästig, wie eine früher einmal entwickelte Routine zu finden, und raten zu müssen, was sie wohl macht.

Obwohl es vielleicht ein wenig kleinkariert erscheint, sollte wirklich die Funktionsweise jeder Routine genau beschrieben werden, es lohnt sich.

Außerdem empfiehlt es sich, in einem Abschnitt am Anfang des Programms eine Dokumentation einzufügen. Diese sollte Copyright-Vermerke, eine Liste der vorgenommenen Änderungen (mit Datum) sowie eine Beschreibung sämtlicher Variablen des Programms enthalten.

Techniken der strukturierten Programmierung

Ein wesentliches Merkmal eines strukturierten Programms ist das Einrücken von zusammenhängenden Code-Abschnitten. Hierdurch wird schon optisch eine Struktur des Programms erkennbar.

Dies betrifft vor allem den Inhalt von Schleifen, aber auch andere Kontrollstrukturen.

Der Blitz2 Editor bietet zwei verschiedene Möglichkeiten Programmzeilen einzurücken. Um einzelne Zeilen einzurücken kann die *TAB*-Taste verwendet. Mit der *BLOCK TAB* Funktion (RAmiga + [und RAmiga +]) können ganze Bereiche gemeinsam eingerückt werden. Die Positionen der Tabulatorsprünge können im Menü *DEFAULTS* eingestellt werden.

Die Tastenkombination *SHIFT*-PfeilLinks dient dazu, den Cursor in die selbe Spalte wie die darüberliegende Zeile zu positionieren.

Modularisierung

Eine sorgfältige Konzeptplanung ist das wichtigste bei der Software-Entwicklung. Diese führt zu einer Zerlegung des Projekts in kleinere Teile, nur so lassen sich albtraumartige Spaghetti-Programme vermeiden.

Nach der Entscheidung darüber, wie das Programm unterteilt werden soll, empfiehlt es sich, mit den schwierigsten Teilen zu beginnen und diese auszutesten, solange das Programm noch klein und überschaubar ist. Denn je größer das Programm ist, desto langwieriger wird die Fehlersuche.

Das gleiche gilt für die Zeit, die man auf den Compiler warten muß, wenn man Fehler sucht oder kleine Änderungen vorgenommen hat und deswegen das gesamte Programm neu compilieren muß. Hier ist es empfehlenswert, den in Frage kommenden Abschnitt aus dem Hauptprogramm herauszunehmen und in einem kleineren Testprogramm auszutesten.

Einige Dinge sollten bei der Entwicklung von Routinen beachtet werden:

- Eine Routine muß mit allen möglichen Situationen fertig werden.
- Der Programmcode sollte effizient sein.
- Modularisierung, d.h. eine Routine kehrt immer dahin zurück, von wo aus sie aufgerufen wurde.
- Ausführlich dokumentieren.
- Besonders den Gebrauch von globalen Variablen und Feldern kommentieren.
- Alle Routinen müssen „wasserdicht“ sein, d.h. sie dürfen nicht abstürzen, wenn sie mit den falschen Parametern gerufen werden.
- Programmabschnitte einrücken und die Zeilenlänge begrenzen, so daß der Code lesbar bleibt.

Nebenbei...

Neben der Kommentierung der einzelnen Routinen im Programm selbst, ist es nützlich, sich eine handgeschriebene Stichwortliste anzulegen, auf der z.B. notiert wird, welche Variablen in mehreren Routinen verwendet werden und welche Routinen noch überarbeitet werden müssen.

Eine solche Liste dient auch dazu, Probleme im voraus zu erkennen.

Einer der größten Fehler beim Programmieren ist es, mit einer Routine zu beginnen, die zum eigenen Funktionieren die Existenz mehrerer anderer Routinen voraussetzt. Es sollten immer zunächst solche Routinen entwickelt werden, die unabhängig getestet werden können. So vermeidet man Situationen wie diese: man hat gerade 5 neue ungetestete Routinen eingebunden und nun muß man versuchen, einen Fehler zu finden, der in jeder dieser Routinen aufgetreten sein kann. Modularität ist auch bei der Entwicklung eines Programmes der effektivste Weg.

Lesbarkeit des Programms

Die Lesbarkeit des Programmcodes ist der nächste Punkt auf der Prioritätenliste bei der Programmentwicklung.

Die beiden wesentlichen Aspekte sind: das Einrücken von geschachtelten Anweisungen und die Minimierung der Zeilenlänge.

Das folgende Beispiel zeigt das Einrücken von verschachteltem Code:

```
If ReadFile (0,"phonebook.data")
      FileInput 0
      While NOT Eof(0)
            If AddItem(people())
                  For i=0 To #num-1
                        \info[i]=Edit$(128)
                  Next
            EndIf
      Wend
EndIf
```

Hierdurch wird es möglich, auf dem ersten Blick zu erkennen, welche Anweisungen innerhalb von welcher Struktur ausgeführt werden. Ebenso kann leicht ermittelt werden, ob für jedes If auch ein EndIf oder für jedes While auch ein Wend vorhanden ist, indem einfach in der entsprechenden Spalte nach dem Gegenstück gesucht wird.

Anhang 5

Blitz2 Operatoren

Ein Ausdruck besteht aus Variablen, Konstanten, Operatoren und Funktionen. Operatoren sind z.B. das Plus- oder Minuszeichen.

Operatoren erzeugen ein Ergebnis unter Verwendung entweder nur des Operanden auf der rechten Seite:

```
a=NOT 5
```

oder der Operanden auf der rechten und linken Seite des Operators:

```
a=5+2
```

Ein Ausdruck kann aus mehreren Operatoren bestehen:

```
a=5*6+3
```

Natürlich hängt das Ergebnis von der Reihenfolge ab, in der die Operationen ausgeführt werden: wird die Multiplikation zuerst ausgeführt, ergibt sich 33, wird die Addition zuerst ausgeführt, also `5*(6+3)` ergibt sich 45.

In Blitz2 ist genau festgelegt, welche Operatoren vorrangig vor anderen ausgeführt werden. Da z.B. die Multiplikation immer vor der Addition ausgeführt wird, sind die folgenden beiden Zeilen gleichwertig:

```
a=5*6+3
a=3+5*6
```

Um die Präzedenz der Operatoren zu umgehen, werden diejenigen Ausdrücke, die zuerst ausgewertet werden sollen, eingeklammert:

```
a=5*(6+3)
```

Die folgende Liste enthält alle Blitz2 Operatoren, gruppiert nach ihrer Priorität (RHS= rechte Seite, LHS= linke Seite). Operatoren in der selben Gruppe haben die gleiche Priorität, die Priorität der Gruppen nimmt nach unten hin ab. Werden zwei Operatoren der selben Priorität in einem Ausdruck verwendet, werden sie in der Reihenfolge ihres Auftretens ausgewertet.

`NOT`	RHS logisch negiert
`-`	RHS arithmetisch negiert
`BITSET`	LHS mit RHS Bits gesetzt
`BITCLR`	LHS mit RHS Bits gelöscht
`BITCHG`	LHS mit RHS Bits gewechselt
`BITTST`	wahr wenn LHS Bits von RHS gesetzt sind
`^`	LHS hoch RHS
`LSL`	LHS logisch um RHS Stellen nach links verschoben
`ASL`	LHS arithmetisch um RHS Stellen nach links verschoben
`LSR`	LHS logisch um RHS Stellen nach rechts verschoben
`ASR`	LHS arithmetisch um RHS Stellen nach rechts verschoben
`&`	logische UND-Verknüpfung zwischen LHS und RHS
`|`	logische ODER-Verknüpfung zwischen LHS und RHS

`*`	LHS multipliziert mit RHS
`/`	LHS dividiert durch RHS
`+`	LHS addiert zu RHS
`-`	RHS subtrahiert von LHS
`=`	wahr wenn LHS gleich RHS
`<>`	wahr wenn LHS ungleich RHS
`<`	wahr wenn LHS kleiner RHS
`>`	wahr wenn LHS größer RHS
`<=`	wahr wenn LHS kleiner oder gleich RHS
`>=`	wahr wenn LHS größer oder gleich RHS
`AND`	logische UND-Verknüpfung zwischen LHS und RHS
`OR`	logische ODER-Verknüpfung zwischen LHS und RHS

Boolsche Operatoren
Die Boolsche Algebra arbeitet mit nur zwei Werten „*wahr*“ und „*falsch*“. In Blitz2 wird der Wert falsch durch *0* repräsentiert, wahr wird durch *-1* repräsentiert. Die Vergleichs-Operatoren `=`, `<>`, `<=`, `=>`, `<` und `>` erzeugen alle ein Boolsches Ergebnis, also *wahr* oder *falsch*.

Die Anweisung `NPrint 2=2` gibt den Wert *-1* aus, da das Ergebnis der Operation 2=2 wahr ist. Die Operatoren `AND`, `OR` und `NOT` können ebenfalls als Boolsche Operatoren verwendet werden.

`Nprint 2=2 AND 5=6` liefert das Ergebnis 0, da nicht beide Bedingungen erfüllt sind. Der Operator `OR` liefert wahr, sobald eine der beiden Bedingungen erfüllt ist, der Operator `NOT` liefert wahr, wenn der Operand falsch ist und umgekehrt.

Binäre Operatoren
Einige der Blitz2 Operatoren arbeiten mit binärer Arithmetik.

Diese Operationen sind besonders schnell, da sie direkt den Prozessor-Instruktionen entsprechen.

Im Binärsystem werden alle Zahlen durch eine Folge von Einsen und Nullen repräsentiert, die „Bits“ genannt werde. Ein „Byte“ besteht aus acht solchen Bits, ein „Wort“ (Word) aus 16 und ein „Doppelwort“ oder „Langwort“ (Longword) aus 32 Bits.

Eine eingehende Erläuterung der Funktionsweise der binären Operatoren findet sich in jeder Beschreibung des 6800 Microprozessors.