A Guide to Blitz Basic

Neil Wright, Mickley 1996

version 1.1

ALL F1 DISKS SHOULD CARRY THE OFFICIAL F1 LABEL
IF YOU HAVE A MULTI DISK PROGRAM THEN ALL THE DISKS IN
THE SET SHOULD BE LABELLED IF NOT PLEASE CONTACT
F1 SOFTWARE ON 01709 888127 OR WRITE TO
1 LOWER MILL CLOSE GOLDTHORPE ROTHERHAM S YORKS S63 9BY

SEND DISK + STAMP FOR FREE CAT DISK.

SCREEN GRAFIX MALCOLM LAVERY (C)1994

PRESS RIGHT MOUSE BUTTON TO CONTINUE

An AmigaGuide conversion from F1 Licenceware disks F1-136A and F1-136B.

Credits

I should like to take this opportunity of thanking those people who helped in the writing of this guide: Jonathan Rutherford who gave valuable comments; Mark Sims and Noel Baldacchino for their programs and suggestions; Kevin Winspear for editing the text into AmigaGuide format. Special thanks must go to my parents, Tony and Barbara, for purchasing the equipment used to create this guide and for proof-reading. Finally I am grateful to Acid Software, the purveyors of quality software on the Amiga.

Neil Wright, Mickley 1996

Preface

This guide is written for new and experienced users of Blitz Basic 2, the revolutionary BASIC language for the Amiga. A GUIDE TO BLITZ BASIC is designed as a complete reference guide, rather than a step-by-step programming tutorial; it will rapidly become one of your most valuable reference works. The guide begins with the basics of programming and by the time you reach its end you will have been introduced to the main features of the Blitz Basic language.

It contains an explanation of every single Blitz command, gives a useful example in each case, and tells you about known bugs and how to work round them. The heart of the guide is a self-instructional Blitz Basic course, taking the reader through basic programming concepts, math commands, graphics, music and sound effects. Shapes, sprites and Intuition are all covered in detail.

You will find the guide most pleasurable if you work through the examples as you read it. Programming is primarily a practical activity and you are encouraged at times to increase understanding of Blitz Basic by creating your own programs.

If you are learning to program Blitz as a hobby you will find it absorbing and intellectually challenging. Even the most inexperienced user will soon develop an appetite for Blitz. After all, programming is fun.

- Disk 2 of the guide contains all of the examples in Blitz Basic format, together with several useful programs and game demos.

Contents

Chapter 1 : The Basics

A couple of years ago the computer press speculated on Commodore's expected domination of the home computer market and the continuing success of the Amiga. Events in 1994 confirmed both, though no-one could have anticipated the eccentric mishandling of the CD32 console and the subsequent caution with which the machine was to be regarded by manufacturers and buyers alike.

At the back end of 1994 it was still uncertain whether the Amiga - and the CD32 especially - would sell in quantities hoped for by Commodore, and which would justify large investment by software houses. They didn't, and the mighty Commodore was destroyed.

However, it hasn't been all doom and gloom. The advent of high-level programming languages, such as Blitz Basic 2, is just one sign amongst many that software development on the Amiga is far from dead.

1.1 Welcome to Blitz Basic

BASIC stands for Beginners All-Purpose Symbolic Instruction Code. It uses an easily grasped mixture of English, numbers, strings, arithmetic signs and parameters which will enable you to start programming without having to learn a daunting low-level language such as Assembly Language. BASIC is a high-level language which was first devised for education purposes only, but during recent years it has undergone many improvements and is now widely used throughout the Amiga world in the form of Blitz Basic 2.

A few years ago Blitz Basic was a breakthrough, the first programming language that ran anywhere near as fast as Assembly Language (with the obvious exception of the C language). It was developed to run solely on the Amiga - an A500 at the time - and some of the peculiarities of that machine have been enshrined in the language ever since.

There were a number of pretenders to Blitz Basic's crown, including AMOS, GFA Basic and HiSoft Basic, but it built up a large following and went through several versions and revisions - like the Amiga, but more slowly. Blitz Basic 2 is currently up to version 1.9, the version covered in this guide, although the information contained herein is relevant in part to all versions of Blitz.

In its relatively short lifetime, Blitz Basic has established itself as the most powerful BASIC dialect on the Amiga. It is certainly a highly satisfactory package for the budding Amiga programmer - this is indicated by the abundance of Blitz-created software in the Public Domain.

Blitz Basic is immensely powerful but does not welcome the novice. That is not because the program is badly implemented - far from it - but because the documentation that accompanies the software is poor and over-complicated. Together with Blitz Basic, this guide will help you unlock the power of your Amiga! Here goes...

1.2 Using this guide

The following chapters provide a thorough and comprehensive index of all the Blitz Basic tokens, as well as a valuable amount of reference material for using Blitz Basic 2.

The commands are arranged in relevant chapters and each description follows an identical format, for ease of reference. After the command name the operating modes are given and they are followed by a brief explanation of the command and the command syntax. For example:

PRINT

```
Mode(s):   Amiga/Blitz
Statement: print items on screen
Syntax:    Print EXPRESSION
```

This is followed by a fuller explanation and, where appropriate, an example of the command's use.

The following conventions are used in the syntax descriptions:

- Command parameters are in capitals
- Square brackets indicate optional parameters [ ]
- Three dots indicate that more parameters of the same format may be added as necessary (...)

If you are already familiar with the Blitz Basic 2 instruction set, be sure to read through the chapters for any information that you may not know. You may be pleasantly surprised!

1.3 Basic programming concepts

The Blitz Basic 2 instruction set consists of a number of reserved keywords which perform a specific task. It includes the names of all Blitz Basic statements, functions, commands and operators. Examples include PRINT, EDIT$, WAITEVENT and <>.

Reserved words can be entered in either uppercase or lowercase, and Blitz Basic will automatically highlight and format the keyword. You should always separate Blitz reserved keywords from parameters, data, or other elements of a command with spaces. This lowers the risk of Blitz Basic not recognising a token name.

1.3.1 Functions, statements & commands

The Blitz Basic 2 instruction set comes in three different flavours: functions, statements and commands.

Functions are Blitz Basic tokens that require parameters in parentheses, and return a value:

```
; *** Functions example
; *** Filename - Functions.bb2

N=Abs(-10)
MouseWait
End
```

Statements are Blitz Basic tokens that only perform an action but do not return a value. Their arguments do not require parentheses:

```
; *** Statements example
; *** Filename - Statements.bb2

NPrint "Blitz Basic 2"
```

```
MouseWait
End
```

Commands are Blitz Basic tokens that can be used as either a function or a statement:

```
; *** Commands example
; *** Filename - Commands.bb2

ev.l=WaitEvent ; *** As a function
Waitevent      ; *** As a statement
MouseWait
End
```

1.4 Amiga Vs Blitz

Blitz Basic runs under two modes, namely Amiga mode and Blitz mode, and some of its commands are limited in the mode under which they can run. Although the Amiga's Operating System is very powerful, it often gets in the way of games programmers and slows the machine down. Blitz mode chucks the Operating System out of the window, so that Blitz Basic can talk directly to the Amiga's hardware. Blitz mode programs run extremely fast, and smooth scrolling and dual playfield displays can be created.

However, all Blitz reserved keywords are restricted in that they can operate under Amiga mode, OR Blitz mode, OR both (the operating modes for all reserved keywords are given in this guide).

The following commands (commonly known as directives) are used to temporarily alter the Blitz Basic operating mode.

AMIGA

```
Mode(s):   Amiga/Blitz
Directive: enter Amiga mode
Syntax:    AMIGA
```

The AMIGA directive is used to enter Amiga mode and to return to the Intuition environment. This is the default Blitz Basic operating mode:

```
; *** AMIGA example
; *** Filename - AMIGA.bb2

; *** Enter Blitz mode
BLITZ
; *** Create Blitz mode display
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
; *** Output some text
```

```
NPrint "Blitz mode"
VWait 100
; *** Enter Amiga mode
AMIGA
; *** Output some more text
DefaultOutput
NPrint "Amiga mode"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

BLITZ

```
Mode(s):   Amiga/Blitz
Directive: enter Blitz mode
Syntax:    BLITZ
```

The BLITZ directive is used to enter Blitz mode. Any further commands which require the presence of the Operating System (such as the File access, Window and Gadget commands) will become temporarily unavailable. File access especially should not occur directly before you enter Blitz mode. To ensure that this is the case, after file access insert the following line before executing the BLITZ directive:

```
VWait 100
```

Blitz mode is not a permanent state. Once your program has finished executing, Blitz Basic returns to Amiga mode. For example:

```
; *** BLITZ example
; *** Filename - BLITZ.bb2

; *** Enter Blitz mode
BLITZ
; *** Create Blitz mode display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

QAMIGA

```
Mode(s):   Amiga/Blitz
Directive: enter Quick Amiga mode
Syntax:    QAMIGA
```

The QAMIGA directive is used to enter Quick Amiga mode. Quick Amiga mode is similar to Amiga mode, however the current display is unaffected (i.e. you are not returned to the Intuition environment). This allows you to jump into Amiga mode without having to corrupt a Blitz mode display. Here's an example:

```
; *** QAMIGA example
; *** Filename - QAMIGA.bb2

; *** Enter Blitz mode
BLITZ
; *** Create Blitz mode display
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
; *** Output some text
NPrint "Blitz mode"
VWait 100
; *** Enter Quick Amiga mode
QAMIGA
; *** Output some more text
NPrint "QAmiga mode (same display)"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.5 Label Definitions

Alphanumeric labels can consist of letters, special characters, or numbers. However, they must begin with an alphabetical character. This allows the use of mnemonic labels to make your program code easier to understand.

For example, the following labels are valid:

```
Bob:
BOB:
A100:
_Print:
```

However, the following label names are not allowed:

```
1:      ; *** begins with a number, not a letter
101:    ; *** begins with a number, not a letter
Print:  ; *** Blitz Basic reserved keyword
```

Capital label names are treated differently to lowercase label names. For example, Bob: and BOB: are recognised as two different labels by Blitz Basic.

1.5.1 Restrictions

Alphanumeric labels are distinguished from variables by a terminating colon (:) - a legal label cannot have a space between the name and the colon. When you refer to a label in a GOSUB or GOTO or other control structure, do not include the colon as part of the label name.

You cannot use any Blitz Basic reserved keyword as an alphanumeric label, as Blitz Basic will generate an error.

1.6 Variables

Variables represent values that are used in a program. In Blitz Basic there are two types of variable: numeric and string. A numeric variable can only be assigned a value that is a number:

```
; *** Variables example 1 ** Filename - Variable1.bb2

; *** Define numeric variable
A=1
; *** Output contents of variable
NPrint A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

A string variable can only be assigned a character string value:

```
; *** Variables example 2 ** Filename - Variable2.bb2

; *** Define string variable
A$="Blitz Basic"
; *** Output contents of variable
NPrint A$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

You can assign a value to a variable, or it can be assigned as the result of calculations in the program - this is known as an expression. Before a variable is assigned a value, its value is zero (numeric variables) or null (string variables).

While a variable name cannot be a reserved keyword, a reserved keyword embedded in a variable name is allowed:

```
; *** Variables example 3
; *** Filename - Variable3.bb2

; *** Define numeric variable
APrint=10
; *** Output contents of variable
NPrint APrint
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LET

```
Mode(s):   Amiga/Blitz
Statement: assign a value to a variable
Syntax:    Let VARIABLE=EXPRESSION
```

LET is an optional statement which is used to assign a value to a variable. For example:

```
; *** Let example
; *** Filename - Let1.bb2

; *** Define numeric variable
Let A=1
; *** Output contents of variable
NPrint A
; *** Define numeric variable
A=1
; *** Output contents of variable
NPrint A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Here are some more examples of LET:

```
; *** Two Let
; *** Filename - Let2.bb2

Let A=180   ; *** Load variable A with 180
Let A=B*10  ; *** Load ten lots of variable B into A
Let B+1     ; *** Increase B by 1
Let B-1     ; *** Decrease B by 1
Let C*10    ; *** Multiply C by 10
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

EXCHANGE

```
Mode(s):   Amiga/Blitz
Statement: swap the contents of two variables
Syntax:    Exchange A,B
```

This useful little statement swaps the contents of two variables of the same type (i.e. A is assigned the value of B and B, the value of A). For example:

```
; *** Exchange example
; *** Filename - Exchange.bb2

NUM=10
; *** Dimension an array
Dim RANDOM(NUM)
; *** Generate NUM (default is 10) numbers
For A=1 To NUM
  RANDOM(A)=A
Next A
Repeat
  Repeat
    ; *** Generate some random numbers
    B=Rnd(NUM)+1
  Until B>0
  ; *** Swap variables
  Exchange RANDOM(B),RANDOM(NUM)
  Let C+1
Until C=NUM
; *** Output random numbers
For T=1 To NUM
  NPrint RANDOM(T)
Next T
```

```
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.7 Numeric types

Blitz Basic currently supports six different types of variable: five numeric types with different ranges and accuracies for numeric data, and one string type ($) for character strings (we'll take a look at the string type later on).

Table 1.1 : Numeric types

```
Type  Suffix Range          Accuracy Bytes Example
==========================================================
Byte  .b     +/- 128        Integer  1     Neil.b=125
Word  .w     +/- 32768      Integer  2     Dan.w=30000
Long  .l     +/- 2147483648 Integer  4     Jon.l=$dff000
Quick .q     +/- 32768.0000 1/65536  2     Richard.q=500/7
Float .f     +/- 9e18       1/10e18  4     Craig.f=4e7
```

To assign a type to a variable simply add the relevant suffix from the above table to the variable name:

```
; *** Blitz Basic types
; *** Filename - Types.bb2

; *** Define numeric variables
BYTE.b=126
WORD.w=32767
LONG.l=3200000
QUICK.q=3.1415
FLOAT.f=3e8
; *** Output numeric variables
NPrint BYTE
NPrint WORD
NPrint LONG
NPrint QUICK
NPrint FLOAT
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

If no suffix is used in the first reference of a variable then Blitz Basic will assign that variable with the default type. The default type is quick, however the DEFTYPE statement can be used to change this.

DEFTYPE

```
Mode(s):   Amiga/Blitz
Statement: declare a list of variables as a particular type
Syntax:    DEFTYPE.TYPE [VARIABLE[,VARIABLE2,...]
```

The DEFTYPE statement has two main uses. It can change the default type and it can also be used to declare a list of variables as being of a particular type (the default type is not affected). In this case, the optional VARIABLE parameters must be included. Here is an example:

```
; *** DEFTYPE example
; *** Filename - DEFTYPE.bb2

A=Pi
; *** A is a quick
NPrint A
; *** Set default type to word
DEFTYPE.w
NPrint Pi
; *** Declare variables A and B as quicks
DEFTYPE.q A,B
A=Pi
B=Sqr(Pi)
NPrint A
NPrint B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SIZEOF

```
Mode(s):  Amiga/Blitz
Function: return amount of memory a variable takes up
Syntax:   s=SizeOf.TYPE[,PATH]
```

This function returns the amount of memory, in bytes, that a variable type takes up. If the optional PATH parameter is included then the offset from the start of the type, to the specified entry, is returned. For example:

```
; *** SizeOf example
; *** Filename - SizeOf.bb2

; *** NewType definition
NEWTYPE.NAME
```

```
  A.l
  B.w
  C.q
End NEWTYPE
; *** Return size of NewType
NPrint SizeOf.NAME
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.7.1 Manipulating quick numbers

As has been explained, the quick type is a fixed point type, with an accuracy of four decimal places. Quick numbers can be manipulated with the following functions.

QLIMIT

```
Mode(s):  Amiga/Blitz
Function: limit the range of a quick number
Syntax:   QLimit(QUICK,LOW,HIGH)
```

Use the QLIMIT function to limit the range of a quick number. If QUICK is greater than or equal to LOW, and less or equal to HIGH, then the value of QUICK is returned. If QUICK is less than LOW then LOW is returned. Conversely, if QUICK is greater than HIGH then HIGH is returned. For example:

```
; *** QLimit example
; *** Filename - QLimit.bb2

NPrint QLimit(100,0,90) ; *** Returns 90
NPrint QLimit(90,95,100) ; *** Returns 95
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

QWRAP

```
Mode(s):  Amiga/Blitz
Function: wrap the result of a quick expression
Syntax:   QWrap(QUICK,LOW,HIGH)
```

QWRAP wraps the result of the quick expression if QUICK is greater than or equal to HIGH, or less than LOW. If QUICK is less than LOW then QUICK-LOW+HIGH is returned. If QUICK is greater than or equal to HIGH then QUICK-HIGH+LOW is returned. Here are some examples:

```
; *** QWrap example
; *** Filename - QWrap.bb2

NPrint QWrap(-10,0,320) ; *** Returns 310
NPrint QWrap(100,0,90) ; *** Returns 10
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.8 NewTypes

In addition to the six primitive types available, programmers can also create their own custom types, or NewTypes. A NewType is a collection of fields, similar to a database or C structure, which enables you to group together relevant fields in one variable type.

NEWTYPE

```
Mode(s):   Amiga/Blitz
Statement: begin a NewType definition
Syntax:    NEWTYPE .NAME
```

END NEWTYPE

```
Mode(s):   Amiga/Blitz
Statement: end a NewType definition
Syntax:    End NEWTYPE
```

NEWTYPE must be followed by a list of fields, separated by colons and/or newlines:

```
NEWTYPE .NAME
  X.w
  Y.w
  SPEED.w
End NEWTYPE
```

Once a NewType is defined, variables are assigned the new type by using a suffix of .NAME:

```
A.NAME
```

Which would assign the contents of the "NAME" NewType to the "A" variable.

1.8.1 NewType fields

When defining a NewType structure, field names without a suffix will be assigned the type of the previous field:

```
NEWTYPE .NAME
  X.l
  Y
  SPEED
End NEWTYPE
```

In the above example the X field is assigned the long type, so the Y and SPEED fields are assigned the same type.

Individual fields within a NewType variable are accessed and assigned using the "\" character:

```
NEWTYPE .NAME
  X.w
  Y.w
  SPEED.w
End NEWTYPE
A.NAME\X=10
NPrint A\X
MouseWait
End
```

Which would assign the value 10 to the X field.

To assign values to all of the fields at once, separate the values with commas:

```
NEWTYPE .NAME
  X.w
  Y.w
  SPEED.w
End NEWTYPE
A.NAME\X=10,20,30
NPrint A\X
```

Which would assign the values 10, 20 and 30 to the X, Y and SPEED fields respectively.

1.8.2 Restrictions

References to string fields do not require the $ or .s suffix to be present. The following example will generate an error:

```
NEWTYPE .NAME
  NAME$
  AGE.q
End NEWTYPE
A.NAME\NAME$="Neil Wright"
NPrint A\NAME$
MouseWait
End
```

This is the correct procedure:

```
NEWTYPE .NAME
  NAME$
  AGE.q
End NEWTYPE
A.NAME\NAME="Neil Wright"
NPrint A\NAME
MouseWait
End
```

1.8.3 NewType in action

Once you have gained an understanding of how NewTypes are created, the next step is to see how they are used within a Blitz Basic program. Here is full example:

```
; *** NEWTYPE example
; *** Filename - NEWTYPE.bb2

; *** Create NewType with three fields
NEWTYPE.NAME
  A.l
  B.w
  C.q
End NEWTYPE
; *** Assign three values to the three fields
A.NAME\A=10,20,30
; *** Output the contents of the fields
NPrint A\A
NPrint A\B
NPrint A\C
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.9 Constants

A constant is a values which is defined by the programmer, but does not change during program execution. Constants are faster than variables and do not consume any memory. However, the following must be obeyed when using constants:

- Constants can only hold integer values
- Constants can be used in assembler
- Constants can be used in conditional compiling evaluation

A constant is defined by adding the hash symbol (#) before a variable name. For example, #X=100 means that the #X variable is a constant, and will always be equal to 100. This allows the Blitz programmer to replace meaningless numbers with mnemonic constants:

```
; *** Constants example
; *** Filename - Constants.bb2

; *** Define constants
#WIDTH=320
#HEIGHT=256
#DEPTH=3

; *** Create Blitz mode display using constants
BLITZ
BitMap 0,#WIDTH,#HEIGHT,#DEPTH
Slice 0,44,#DEPTH
Show 0
BitMapOutput 0
; *** Output contants
NPrint "Width = ",#WIDTH
NPrint "Height = ",#HEIGHT
NPrint "Depth = ",#DEPTH
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.10 Strings

A string variable is one which contains text, rather than numbers. Strings are surrounded by quotation marks and all string names must end with the dollar ($) character. They can comprise of characters, numbers or spaces. The example below creates a new string (A$) and stuffs it with the contents of the subsequent quote marks:

```
; *** Strings example
; *** Filename - Strings1.bb2

; *** Define a numeric variable
A$="Blitz BASIC 2"
```

```
; *** Output variable
Print A$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

MAXLEN

```
Mode(s):   Amiga/Blitz
Statement: define maximum length of string variable
Syntax:    MaxLen "STRING"=EXPRESSION
```

The MAXLEN statement is used to define the maximum length of a string variable. EXPRESSION specifies the maximum number of characters for the string. This is only necessary when using the Blitz Basic commands which require this definition (FILEREQUEST$ and FIELDS). Try the following example:

```
; *** MaxLen example
; *** Filename - MaxLen.bb2

; *** Open a hi-res screen
Screen 0,3+8
; *** Set maximum length of variables
MaxLen PATH$=160
MaxLen FILENAME$=64
; *** Create a file requester
F$=FileRequest$("File requester",PATH$,FILENAME$)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Blitz Basic's string functions are extremely powerful, which is why we have devoted the whole of Chapter 2 to them.

1.11 Blitz Basic operators

Operators perform mathematical or logical operations on values. When several operators are used within the same program statement, they are processed in a specific order. This order is dependent on the operator list, or hierarchy. The operators found at the top of this list are processed first. If the operators are of the same level, the leftmost one is executed first, the rightmost last:

Table 1.2 : Blitz Basic operators

Operator	Description	Example
NOT	Logical NOT	NOT A
BITSET	A with B bit set	A BitSet B
BITCLR	A with B bit cleared	A BitClr B
BITCHG	A with B bit changed	A BitChg B
BITTST	True if A bit of B set	A BitTst B
^	Exponentiation	A^B
LSL	A left B times (logical)	A LSL B
ASL	A left B times (arithmetical)	A ASL B
LSR	A right B times (logical)	A LSR B
ASR	A right B times (arithmetical)	A ASR B
&	Logical AND	A&B
|	Logical OR	A|B
*	Multiply	A*B
/	Divide	A/B
+	Add	A+B
-	Subtract	A-B
=	Equal	A=1
<>	Unequal	A<>B
<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal to	A<=B
>=	Greater than or equal to	A>=B
AND	Logical AND	A AND B
OR	Logical OR	A OR B

1.11.1 Relational operators

Relational operators are used to compare two values. The result of the comparison is either true (-1) or false (0). This result can then be used to make a decision regarding program execution. The following table lists the relational operators:

Table 1.3 : Relational operators

Operator	Description	Example
=	Equal	A=1
<>	Unequal	A<>B
<	Less than	A<B
>	Greater than	A>B
<=	Less than or equal to	A<=B
>=	Greater than or equal to	A>=B

The "=" operator compares two numerical or character string expressions. When both are equal the logical true is returned, otherwise logical false will be returned:

```
; *** = operator
; *** Filename - =.bb2

A=3
B=3
If A=B Then End
Repeat
Forever
```

The "<", ">", "<=" and ">=" operators serve to compare numerical and string expressions:

```
* A>B is true when A is greater than B
* A<B is true when A is less than B
* A<=B is true then A is less than or equal to B
* A>=B is true when A is greater than or equal to B
```

The "<>" operator determines if two numerical or string expressions are unequal:

```
* A<>B is true when A is unequal to B
```

When arithmetic and relational operators are combined in one expression, the arithmetic operation is always performed first.

1.11.2 Logical operators

Logical operators perform bit manipulation, Boolean operations, or tests on multiple relations. Like relational operators, logical operators can be used to make decisions regarding program execution.

A logical operator returns the result from the combination of true-false operands. The result (in bits) is either true (-1) or false (0).

The Blitz Basic logical operators are NOT (logical complement), AND (conjunction) and OR (disjunction).

For example:

```
; *** Logical operators
; *** Filename - Logic.bb2

NPrint NOT 3     ; *** returns -4
NPrint NOT -4    ; *** returns 3
NPrint 50 AND 40 ; *** returns 32
NPrint 12 AND 11 ; *** returns 8
NPrint 2 OR 1    ; *** returns 3
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

1.12 Using operators with strings

A string expression consists of string constants, string variables, and other string expressions combined by operators. There are two types of string operation: concatenation and relation.

1.12.1 Concatenation

Combining two strings together is called concatenation. The plus (+) operator is used to perform concatenation. Here is an example of the use of the operator:

```
; *** A piece of string
; *** Filename - Strings2.bb2

; *** Define string variables
A$="Blitz "
B$="Basic "
C$="is tops!"
; *** Concatenate strings
Print A$+B$+C$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

In the above example the "+" operator is used join together two strings. Running the example produces the following on the screen:

```
Blitz Basic is tops!
```

Note that the other arithmetic operators (i.e. -, /, *) should not be applied to strings.

1.12.2 Relational operators

Strings can also be compared using the same relational operators that are used with numeric variables (i.e. =, <, >, <>, <= and >=).

With strings, the relational operators compare the ASCII codes of the characters which comprise the string. The ASCII code system assigns a different number to each keyboard character. If all the ASCII codes are the same, the strings are equal. If the ASCII codes differ, the lower code number precedes the higher.

All string constants used in comparison expressions must be enclosed in quotation marks.

Here is an example:

```
; *** Relational operators
; *** Filename - Relation.bb2

; *** Define string variables
A$="A"
B$="B"
C$="Blitz"
D$="Basic"
; *** Evaluate variables
If A$<B$ Then NPrint A$,"<",B$
If B$>A$ Then NPrint B$,">",A$
If C$>D$ Then NPrint C$,">",D$
If C$=C$ Then NPrint C$,"=",C$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.13 Arrays

An array is a list of variables of the same name that are distinguished by subscripts (values that identify each variable or element in the array). Arrays can be made up from any type of variable. Creation of such an array is accomplished by the DIM statement.

DIM

```
Mode(s):   Amiga/Blitz
Statement: dimension an array
Syntax:    Dim ARRAY_NAME(DIMENSION LIST)
Syntax 2:  Dim List ARRAY_NAME(DIMENSION LIST)
```

The DIM statement is used to dimension (set up) an array of a given number of numeric or string variables. In numeric arrays, DIM is followed by a single letter or word that names the array, and one or more numeric values (dimensions) separated by commas. String arrays are created in the same way, however a single letter or word followed by a ($) is used for the array name. Here is an example:

```
; *** Dim example
; *** Filename - Dim.bb2

; *** Dimension array
Dim A(20)
; *** Define array contents
For B=1 To 20
  A(B)=Int(Rnd(100))
Next B
```

```
; *** Print array contents
For C=1 To 20
  NPrint A(C)
Next C
; *** Wait for mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.13.1 List arrays

The optional List parameter, if included, denotes a List array. List arrays differ from normal arrays in that Blitz Basic keeps an internal count of how many elements are stored in the List and an internal pointer to the current element within the List. List arrays are restricted in size to one dimension:

```
; *** Dim example 2
; *** Filename - Dim2.bb2

; *** Dimension List array
Dim List A(20)
; *** Define array contents
For B=1 To 20
  A(B)=Int(Rnd(100))
Next B
; *** Output List array contents
For C=1 To 20
  NPrint A(C)
Next C
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

RESETLIST

```
Mode(s):   Amiga/Blitz
Statement: reset List array to first item
Syntax:    ResetList ARRAY()
```

RESETLIST is used to set the current List array element to the first item. This prepares the array for processing with the NEXTITEM statement. For example:

```
; *** ResetList example
; *** Filename - ResetList.bb2

; *** Dimension List array
```

```
Dim List A(10)
; *** Process List array
While AddFirst(A())
  A()=B
  Let B+1
Wend
ResetList A()
; *** Output List array contents
While NextItem(A())
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CLEARLIST

```
Mode(s):   Amiga/Blitz
Statement: clear a List array
Syntax:    ClearList ARRAY()
```

The CLEARLIST statement clears a List array. List arrays are automatically cleared when they are dimensioned. Here is an example:

```
; *** ClearList example
; *** Filename - ClearList.bb2

; *** Dimension List array
Dim List A(10)
; *** Process List array
While AddFirst(A())
  A()=B
  Let B+1
Wend
ClearList A()
ResetList A()
; *** Output List array contents
While NextItem(A())
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ADDFIRST

```
Mode(s):  Amiga/Blitz
Function: insert array item at the beginning of an array List
Syntax:   a=AddFirst [ARRAY()]
```

This function enables you to insert an array item at the beginning of a List array. ADDFIRST returns (-1) if there is enough room in the array to add an element, and (0) if no array element is available. Example:

```
; *** AddFirst example
; *** Filename - AddFirst.bb2

; *** Dimension List array
Dim List A(100)
; *** Process List array
While AddFirst(A())
  A()=B
  Let B+1
Wend
NPrint B," items added"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ADDLAST

```
Mode(s):  Amiga/Blitz
Function: insert array item at the end of an array List
Syntax:   a=AddLast [ARRAY()]
```

The ADDLAST function enables you to insert an array item at the end of a List array. It returns (-1) if there is enough room in the array to add an element, and (0) if no array element is available. For example:

```
; *** AddLast example
; *** Filename - AddLast.bb2

; *** Dimension List array
Dim List A(100)
; *** Process List array
While AddLast(A())
  A()=B
  Let B+1
Wend
```

```
; *** Output List array contents
For C=1 To 100
  NPrint A(C)
Next C
NPrint B," items added"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ADDITEM

```
Mode(s):  Amiga/Blitz
Function: insert array item after current item in array List
Syntax:   a=AddItem [ARRAY()]
```

ADDITEM enables you to insert an array item after a List array's current item. The function returns (-1) and sets the array's "current item" pointer to the item added if there is enough room to add an element, and (0) if no array element is available. For example:

```
; *** AddItem example
; *** Filename - AddItem.bb2

; *** Dimension List array
Dim List A(2)
; *** Process List array
If AddFirst(A()) Then A()=1
If AddItem(A()) Then A()=2
If AddItem(A()) Then A()=3
NPrint "List array is:-"
ResetList A()
; *** Output List array contents
While NextItem(A())
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

KILLITEM

```
Mode(s):   Amiga/Blitz
Statement: remove current item from array List
Syntax:    k=KillItem ARRAY()
```

The KILLITEM statement is used to delete the current item from a List array. The "current item" pointer is then set to the item before the deleted element. Here is an example:

```
; *** KillItem example
; *** Filename -KillItem.bb2

; *** Dimension List array
Dim List A(30)
; *** Process List array
While AddItem(A())
  A()=B
  Let B+1
Wend
ResetList A()
While NextItem(A())
  If A()/2<>Int(A()/2)
    KillItem A()
  EndIf
Wend
ResetList A()
; *** Output List array contents
While NextItem(A())
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

PREVITEM

```
Mode(s):  Amiga/Blitz
Function: set pointer to previous item
Syntax:   p=PrevItem [ARRAY()]
```

This function sets the List array's "current item" pointer to the previous item, allowing for backward processing of a List array. PREVITEM returns (-1) if a previous item is available, and (0) if one is unavailable. Try the following example:

```
; *** PrevItem example
; *** Filename - PrevItem.bb2

; *** Dimension List array
Dim List A(25)
; *** Process List array
While AddLast(A())
  A()=B
  Let B+1
Wend
If LastItem(A())
  ; *** Output List array contents
  Repeat
    NPrint A()
  Until NOT PrevItem(A())
EndIf
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

NEXTITEM

```
Mode(s):  Amiga/Blitz
Function: set pointer to next item
Syntax:   n=NextItem [ARRAY()]
```

The NEXTITEM function sets the List array's "current item" pointer to the next item, allowing for forward processing of a List array. It returns (-1) if the next item is available, and (0) if one is unavailable. Example:

```
; *** NextItem example
; *** Filename - NextItem.bb2

; *** Dimension List array
Dim List A(25)
; *** Process List array
While AddLast(A())
  A()=B
  Let B+1
Wend
ResetList A()
; *** Output List array contents
While NextItem(A())
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FIRSTITEM

```
Mode(s):  Amiga/Blitz
Function: set pointer to first item
Syntax:   f=FirstItem [ARRAY()]
```

FIRSTITEM sets the "current item" pointer in a List array to the first item in the array. The function returns (-1) if there is a first item available, and (0) if there are no items in the List array. For example:

```
; *** FirstItem example
; *** Filename - FirstItem.bb2

; *** Dimension List array
Dim List A(25)
; *** Process List array
While AddFirst(A())
  A()=B
  Let B+1
Wend
If FirstItem(A())
  NPrint "First item = ",A()
EndIf
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LASTITEM

```
Mode(s):  Amiga/Blitz
Function: set pointer to last item
Syntax:   l=LastItem [ARRAY()]
```

LASTITEM sets the "current item" pointer in a List array to the last item in the array. The function returns (-1) if there is a last item available, and (0) if there are no items in the List array. For example:

```
; *** LastItem example
; *** Filename - LastItem.bb2

; *** Dimension List array
Dim List A(25)
; *** Process List array
While AddFirst(A())
  A()=B
  Let B+1
```

```
Wend
If LastItem(A())
  NPrint "Last item = ",A()
EndIf
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

PUSHITEM

```
Mode(s):   Amiga/Blitz
Statement: push pointer to internal stack
Syntax:    PushItem ARRAY()
```

The PUSHITEM statement "pushes" a List array's "current item" pointer onto an internal stack. This pointer can be recalled at a later date by POPITEM. The internal item pointer stack can be pushed 8 times.

POPITEM

```
Mode(s):   Amiga/Blitz
Statement: get pointer from internal stack
Syntax:    PopItem ARRAY()
```

POPITEM retrieves a pushed "current item" pointer from the internal stack. The ARRAY() parameter must be the name of the most recently pushed List array. Here's an example:

```
; *** PushItem/PopItem example
; *** Filename - PopItem.bb2

; *** Dimension List array
Dim List A(10)
; *** Process List array
While AddLast(A())
  A()=B
  Let B+1
Wend
ResetList A()
While NextItem(A())
  If A()=5 Then PushItem A()
Wend
PopItem A()
KillItem A()
ResetList A()
; *** Output List array contents
While NextItem(A())
```

```
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ITEMSTACKSIZE

```
Mode(s):   Amiga/Blitz
Statement: set push stack size
Syntax:    ItemStackSize MAXIMUM
```

This statement defines the maximum number of List array items that may be pushed.

1.13.2 Sorting arrays

If you were creating a database-type application, or a program that required the contents of an array to be in order, then it would be very time consuming to manually sort through the array. Blitz Basic provides four statements which can be used to automatically order an array.

SORT

```
Mode(s):   Amiga/Blitz
Statement: sort a specified array in ascending order (default)
Syntax:    Sort ARRAY()
```

The SORT statement sorts the specified array in ascending order. The direction of the sort may be changed using the SORTUP and SORTDOWN statements (default is ascending). Note that NewType arrays and List arrays cannot be sorted with this statement. Here is a full example:

```
; *** Sort example
; *** Filename - Sort.bb2

; *** Dimension an array
Dim A(10)
; *** Create an array of random numbers
For B=1 To 10
  A(B)=Int(Rnd(100))
  NPrint A(B)
Next B
NPrint ""
; *** Sort the array in ascending order
Sort A()
; *** Output array
For C=1 To 10
  NPrint A(C)
```

```
Next C
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SORTUP

```
Mode(s):   Amiga/Blitz
Statement: force the SORT command to sort into ascending order
Syntax:    SortUp
```

Example:

```
; *** SortUp example
; *** Filename - SortUp.bb2

; *** Dimension an array
Dim A(10)
; *** Create an array of random numbers
For B=1 To 10
  A(B)=Int(Rnd(100))
Next B
; *** Sort array in ascending order
SortUp
Sort A()
; *** Output new array
For C=1 To 10
  NPrint A(C)
Next C
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SORTDOWN

```
Mode(s):   Amiga/Blitz
Statement: force the SORT command to sort into descending order
Syntax:    SortDown
```

Example:

```
; *** SortDown example
; *** Filename - SortDown.bb2

; *** Dimension an array
Dim A(10)
; *** Create an array of random numbers
For B=1 To 10
  A(B)=Int(Rnd(100))
Next B
; *** Sort array in descending order
SortDown
Sort A()
; *** Output new array
For C=1 To 10
  NPrint A(C)
Next C
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SORTLIST

```
Mode(s):   Amiga/Blitz
Statement: rearrange the elements in a linked list
Syntax:    SortList ARRAY()
```

The SORTLIST statement is used to rearrange the order of elements in a Blitz Basic linked list. The order in which the items are sorted depends on the first field of the linked list type, which must be a single integer word:

```
; *** SortList example
; *** Filename - SortList.bb2

; *** Dimension a List array
Dim List A(10)
; *** Create a List array of random numbers
```

```
While AddLast(A())
  A()=Int(Rnd(100))
  Let B+1
Wend
ResetList A()
; *** Sort List array
SortList A(),0
; *** Output new array
While NextItem(A())
  NPrint A()
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

1.14 Program control

The following statements are used to control Blitz Basic program execution.

END

```
Mode(s):   Amiga/Blitz
Statement: end the current program
Syntax:    End
```

This statement serves to end the current program. Program execution may not be continued. For example:

```
; *** End example
; *** Filename - End.bb2

NPrint "Press left mouse button to return to editor"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

STOP

```
Mode(s):   Amiga/Blitz
Statement: interrupt the current program
Syntax:    Stop
```

The STOP statement interrupts the current program. Program execution may be resumed using the CONT statement:

```
; *** Stop example
; *** Filename - Stop.bb2

A=Int(Rnd(5))
NPrint "Press left mouse button to stop"
; *** Wait for a mouse click
MouseWait
; *** Stop program in its tracks
Stop
```

CONT

```
Mode(s):   Amiga/Blitz
Statement: continue current program
Syntax:    Cont [N]
```

This statement is only available in direct mode. CONT resumes program execution from the instruction following the STOP statement. The optional N parameter can be used to ignore a specified number of STOP statements after a CONT.

1.15 Using data

What is Data? Well, 99% of all programs ever written operate on and use data of one kind or another. Information and data are really one and the same; we enter information into a computer and get out a different type of information at the end of processing. So when the information is inside the computer, we refer to it as data. Large amounts of this data can be stored in your Blitz programs with the DATA statement.

DATA

```
Mode(s):   Amiga/Blitz
Statement: define data items in a program
Syntax:    Data LIST
Syntax 2:  Data .TYPE LIST
```

READ

```
Mode(s):   Amiga/Blitz
Statement: read data into a variable
Syntax:    Read LIST
```

RESTORE

```
Mode(s):   Amiga/Blitz
Statement: set the current READ pointer
Syntax:    Restore PROGRAM_LABEL
```

The DATA statement allows you to store constant values in your programs. A data pointer is associated with the commands DATA and READ. This pointer always points to the next DATA item to be read with the READ statement and is set initially to the first DATA item. The data pointer can be set at a specific DATA line with the RESTORE statement. For this purpose, a label must be set in front of the DATA line and the data pointer set with RESTORE PROGRAM_LABEL. If no label follows the RESTORE statement the data pointer will be set to the very first DATA item in the program. Here is an example:

```
; *** Using Data ** Filename - Data.bb2

; *** Dimension a string array
Dim A$(5)
; *** Return location of program data
Restore MY_DATA
; *** Read data elements into string array
For A=1 To 5
  Read A$(A)
  NPrint A$(A)
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End

; *** Program data
MY_DATA:
Data$ "Blitz","Basic","Is","Truly","Remarkable"

When data is being read into a variable, the .TYPE of the data being read
must match the type of the variable it is being read into:
```

Table 1.4 : Data types

Data	Data Type	Example
Byte	Data.b	Data.b=125
Word	Data.w	Data.w=30000
Long	Data.l	Data.l=$dff000
Quick	Data.q	Data.q=500/7
Floating point	Data.f	Data.f 3.14,1.79
String	Data$	Data$ "Blitz","BASIC"

For example:

```
; *** Using Data 2 ** Filename - Data2.bb2
; *** Read program data into variables
Read A$,B,C.w
; *** Output variable contents
NPrint A$
NPrint B
NPrint C
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End

; *** Program data
Data$ "Blitz" ; *** String type data
Data 39 ; *** Quick type data
Data -10 ; *** Word type data
```

1.16 End-of-Chapter summary

There are three different types of Blitz Basic token: functions, statements and commands.

Functions are Blitz Basic tokens that require parameters in parentheses, and return a value.

Statements are Blitz Basic tokens that only perform an action but do not return a value. Their arguments do not require parentheses.

Commands are Blitz Basic tokens that can be used as either a function or a statement, depending upon whether the arguments were in parentheses or not.

Blitz Basic 2 runs under two modes: Amiga and Blitz. For system-friendly programs use Amiga mode and, for extra speed, throw the operating system out of the window with Blitz mode.

Variables represent values that are used in a program. Blitz Basic supports six different types of variable: five numeric types with different ranges and accuracies for numeric data, and one string type ($) for character strings. Custom types can be created with the NEWTYPE statement.

Constants are values which are defined by the programmer, but do not change during program execution.

A string variable is one which contains text, rather than numbers. Strings are surrounded by quotation marks and all string names must end with the dollar ($) character.

Operators perform mathematical or logical operations on values.

An array is a list of variables of the same name that are distinguished by subscripts (values that identify each variable or element in the array). List arrays are limited in size to one dimension.

Program execution is stopped with the END statement.

Large amounts of data can be stored in your programs with the DATA statement.

Chapter 2 : String Functions

As we have already found out, a string variable is one which contains text, rather than numbers. Strings are surrounded by quotation marks and all string names must end with the dollar ($) character. They can comprise of characters, numbers or even spaces.

In this chapter you will learn how to manipulate strings using the powerful Blitz Basic string functions.

2.1 Strings and roundabouts

Strings can be sliced, diced and chopped up into individual words and letters using LEFT$, RIGHT$ and MID$. These are the most powerful string functions in Blitz Basic.

LEFT$

```
Mode(s):  Amiga/Blitz
Function: return the leftmost characters of a string
Syntax:   destination$=Left$(SOURCE$,NUMBER_OF_CHARACTERS)
```

LEFT$ takes the specified number of characters from a source string, beginning with the first character, and pastes them into a destination string. For example:

```
; *** Left$ example
; *** Filename - Left$.bb2

NPrint "Enter a string:"
; *** Input text string (40 characters maximum)
A$=Edit$(40)
; *** Input number of characters
NPrint "How many characters from left?:"
A=Edit(10)
; *** Grab the specified characters
NPrint Left$(A$,A)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

RIGHT$

```
Mode(s):  Amiga/Blitz
Function: return the rightmost characters of a string
Syntax:   destination$=Right$(SOURCE$,NUMBER_OF_CHARACTERS)
```

The equivalent function for the right-hand side of text strings is the aptly named RIGHT$. RIGHT$ takes the specified number of rightmost characters from a source string and pastes them into a destination string. The following example can be used to generate numbers with preceding zero characters, such as those found in shoot-em-up games and high-score tables:

```
; *** Right$ example
; *** Filename - Right$.bb2

SCORE=1000
; *** Turn variable into a string
S$=Str$(SCORE)
; *** Add zeros
S$=Right$("0000000"+S$,7)
NPrint S$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

MID$

```
Mode(s):  Amiga/Blitz
Function: return number of characters from middle of string
Syntax:   destination$=Mid$(SOURCE$,START[,NUMBER_OF_CHARACTERS])
```

The MID$ function also works along the same lines. It returns the specified number of characters from the middle of a string, starting with character number START. If the optional NUMBER_OF_CHARACTERS parameter is omitted then all characters from START to the end of the string are returned. Here are some examples which demonstrate the correct use of MID$.

Our first example splits up the source string (A$) into its component letters and displays them vertically:

```
; *** Vertical text
; *** Filename - Vertical_Text.bb2

A$="Blitz Basic"
; *** Use Workbench screen for output
WbToScreen 0
; *** Send Workbench to front of display
WBenchToFront_
; *** Attach window to Workbench screen
Window 0,10,50,600,160,$1|$2|$3|$8,"Vertical Text",0,1
For N=1 To Len(A$)
  ; *** Split string up into characters
  NPrint Mid$(A$,N,1)
Next
; *** Wait for a mouse click
```

```
MouseWait
; *** Send Workbench to back of display
WBenchToBack_
; *** Return to Blitz Basic 2 editor
End
```

The second example also splits up a string into letters, but this time the string is displayed horizontally, one character at a time. This results in a "typewriter" effect, although some typists may disagree!:

```
; *** Typewriter
; *** Filename - Typewriter.bb2

; *** Character delay
DELAY=4
; *** Text string to output
TXT$="Blitz Basic is the most versatile BASIC on planet Earth!"
; *** Use Workbench screen for output
WbToScreen 0
; *** Send Workbench to front of display
WBenchToFront_
Window 0,10,50,600,160,$1|$2|$3|$8,"Typewriter",0,1
; *** Character pointer
A=1
For B=0 To Len(TXT$)
  ; *** Split string up into characters
  Print Mid$(TXT$,A,1)
  Let A+1
  ; *** Pause typewriter
  VWait DELAY
Next B
; *** Send Workbench to back of display
WBenchToBack_
; *** Return to Blitz Basic 2 editor
End
```

2.2 Manipulating strings

Just as numbers can be added, subtracted, multiplied and divided, strings can be manipulated with the following functions.

UNLEFT$

```
Mode(s):  Amiga/Blitz
Function: remove a number of rightmost characters from string
Syntax:   new$=UnLeft$(SOURCE$,LENGTH)
```

UNLEFT$ removes the specified number of characters from the end of a string and places the remaining characters into a new string. The LENGTH parameter specifies the number of characters to remove. For example:

```
; *** UnLeft$ example
; *** Filename - UnLeft$.bb2

NPrint "Enter a string:"
; *** Input text string to manipulate
A$=Edit$(40)
; *** Input number of characters
NPrint "How many characters from end?:"
A=Edit(10)
; *** Output new text string
NPrint UnLeft$(A$,A)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

UNRIGHT$

```
Mode(s):  Amiga/Blitz
Function: remove a number of leftmost characters from string
Syntax:   new$=UnRight$(SOURCE$,LENGTH)
```

Newton's third law of motion states thus: "Every action has an equal and opposite reaction.". In Blitz Basic it often seems like every function has an equal and opposite function! UNRIGHT$, as you may have guessed, removes a specified number of characters from the beginning of a string. The LENGTH parameter specifies the number of characters to remove. Here's an example:

```
; *** UnRight$ example
; *** Filename - UnRight$.bb2

; *** Returns "Blitz is best!"
Print UnRight$("AMOSBlitz is best!",4)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LSET$*

```
Mode(s):  Amiga/Blitz
Function: return a string of text of a given length
Syntax:   new$=LSet$(SOURCE$,LENGTH)
```

The LSET$ function returns a string of text of LENGTH characters long. If SOURCE$ is shorter than LENGTH then the right-hand side of the string will be padded with spaces. The string will be truncated if it is longer than LENGTH. Here are some examples:

```
; *** LSet$ example
; *** Filename - LSet$.bb2

; *** Returns "Blitz"
NPrint LSet$("Blitz Basic",5)
; *** Returns "S"
NPrint LSet$("Spaced out!",1)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

RSET$

```
Mode(s):  Amiga/Blitz
Function: return a string of text of a given length
Syntax:   new$=RSet$(SOURCE$,LENGTH)
```

The RSET$ function returns a string of text of LENGTH characters long. If SOURCE$ is shorter than LENGTH then the left-hand side of the string will be padded with spaces. The left-hand side of the string will be truncated if it is longer than LENGTH. Try the following examples:

```
; *** RSet$ example
; *** Filename - RSet$.bb2

; *** Returns "Wright"
NPrint RSet$("Neil Wright",6)
; *** Centres "Richard Irving"
NPrint RSet$("Richard Irving",45)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

STRIPLEAD$

```
Function: remove all leading occurrences of a character
Syntax:   new$=StripLead$(SOURCE$,EXPRESSION)
```

The STRIPLEAD$ function removes all leading occurrences of an ASCII character from a source string. EXPRESSION is the decimal ASCII code value to be removed. For example:

```
; *** StripLead$ example
; *** Filename - StripLead$.bb2

; *** Remove leading B character
; *** (Returns "litz Basic")
Print StripLead$("Blitz Basic",66)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

To remove all trailing occurrences of an ASCII character the equivalent STRIPTRAIL$ function may be used.

TRIPTRAIL$

```
Mode(s):  Amiga/Blitz
Function: remove all trailing occurrences of a character
Syntax:   new$=StripTrail$(SOURCE$,EXPRESSION)
```

Here is an example:

```
; *** StripTrail$ example
; *** Filename - StripTrail$.bb2

A$="There are          "
; *** Remove trailing spaces
Print StripTrail$(A$,32)+" no spaces"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CENTRE$

```
Mode(s):  Amiga/Blitz
Function: return a centred text string
Syntax:   a$=Centre$(SOURCE$,CHARACTERS)
```

The CENTRE$ function returns a centred text string of length CHARACTERS. If SOURCE$ is shorter than the specified number of characters then the resulting string will be padded with spaces. If SOURCE$ is longer then it will be truncated on either side. Try the following examples:

```
; *** Detention centre$
; *** Filename - Centre$.bb2

; *** Returns "tz Ba"
NPrint Centre$("Blitz Basic",5)
; *** Returns " Blitz "
NPrint Centre$("Blitz",7)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Just as dinosaurs can be cloned (or so movie-makers would like us to believe!), so can text strings. Blitz Basic doesn't support DNA directly so you have to make do with the STRING$ function.

STRING$

```
Mode(s):  Amiga/Blitz
Function: create a new string using copies of an old string
Syntax:   new$=String$(SOURCE$,NUMBER)
```

STRING$ will create a new string containing a specified number of copies of a source string. For example:

```
; *** String$ example
; *** Filename - String$.bb2

NPrint ""
NPrint "Input any old garbage:-"
NPrint ""
; *** Input a text string
A$=Edit$(10)
NPrint ""
; *** Multiply string by five
B$=String$(A$,5)
```

```
Print "* 5 = ",B$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

2.3 String searching

Applications such as word processors and programming utilities often have string searching facilities. These work by breaking down and analysing text files until matching strings are found (the search), and replacing text with new strings (the replace). Blitz Basic has two commands for total control over string searching.

2.3.1 Searching for characters in a string

INSTR

```
Mode(s):  Amiga/Blitz
Function: search for occurrences of one string within another
Syntax:   a=Instr(SOURCE$,FIND$[,START])
```

The INSTR function enables you to search for one string within another. If the search is successful then the character position of the first matching character will be returned. If the search is unsuccessful then (0) will be returned.

The optional START parameter allows you to specify the starting character for the search. Values for START may range from zero to the length of the string. Here is an example:

```
; *** String searching
; *** Filename - Instr_Example.bb2

Restore DAT
NPrint "Input a letter or word to search"
; *** Input word to search
SEARCH$=Edit$(40)
; *** Convert to upper case
SEARCH$=UCase$(SEARCH$)
For A=1 To 6
  ; *** Read data statements
  Read A$
  B$=UCase$(A$)
  ; *** Does string exist?
  If Instr(B$,SEARCH$)
    Print A$
    ; *** Wait for a mouse click
    MouseWait
    End
  End If
```

```
Next A
; *** Unsuccessful search
NPrint "No match found"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End

; *** Data to search
DAT:
Data$ "Blitz"
Data$ "AMOS"
Data$ "GFA"
Data$ "Hisoft"
Data$ "Amiga"
Data$ "Spectrum"
```

The CASESENSE statement may be used to determine whether the search is case sensitive.

2.3.2 Replacing characters in a string

REPLACE$

```
Mode(s):  Amiga/Blitz
Function: replace any occurrences of a string with a new string
Syntax:   new$=Replace$(SOURCE$,FIND$,NEW$)
```

REPLACE$ is used to search for one string within another and replace it with a new string. SOURCE$ is the string to be searched, FIND$ is the string to be found and NEW$ is the replacement string. For example:

```
; *** Replace$ example
; *** Filename - Replace$.bb2

; *** Text string to manipulate
A$="AMOS Basic is tops!"
NPrint A$
VWait 50
; *** Replace the word "AMOS" with "Blitz"
A$=Replace$(A$,"AMOS","Blitz")
NPrint A$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

The CASESENSE statement may be used to determine whether the search is case sensitive.

2.3.3 Case sensitivity

Upper and lower case letters are treated as completely different letters by Blitz Basic. For example, the word "natch" is different to "NATCH". This is known as case sensitivity. The INSTR and REPLACE$ commands are, by default, case sensitive.

CASESENSE

```
Mode(s):   Amiga/Blitz
Statement: control the searching mode used by INSTR and REPLACE$
Syntax:    CaseSense On/Off
```

CASESENSE is used to determine whether or not the INSTR and REPLACE$ functions are case sensitive.

In the example below, the string to search must be entered in the correct case (e.g. DOG, cAT, HAmster), otherwise a match will not be found:

```
; *** CaseSense example
; *** Filename - CaseSense.bb2

Dim A$(5)
; *** Read data into memory
Restore DAT
For A=1 To 5
  Read A$(A)
Next A
; *** Case sensitivity on
CaseSense On
NPrint "Input string to search (from data list)"
B$=Edit$(10)
For B=1 To 5
  ; *** Exact match found
  If Instr(A$(B),B$)=1
    NPrint B$," found at location ",B
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  Else
    ; *** No match found
    NPrint B$," not found"
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
Next B

; *** Program data
DAT:
```

```
Data$ "DOG"
Data$ "cAt"
Data$ "HAmster"
Data$ "rAt"
Data$ "WoMaN"
```

This is the same example, but this time without case sensitivity (i.e. the strings can be entered in any case). For example:

```
; *** CaseSense example 2
; *** Filename - CaseSense2.bb2

Dim A$(5)
; *** Read data into memory
Restore DAT
For A=1 To 5
  Read A$(A)
Next A
; *** Case sensitivity off
CaseSense Off
NPrint "Input string to search (from data list)"
B$=Edit$(10)
For B=1 To 5
  ; *** Match found (case ignored)
  If Instr(A$(B),B$)=1
    NPrint B$," found at location ",B
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  Else
    ; *** No match found
    NPrint B$," not found"
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
Next B

; *** Program data
DAT:
Data$ "DOG"
Data$ "cAt"
Data$ "HAmster"
Data$ "rAt"
Data$ "WoMaN"
```

2.4 Converting strings

One facility that we haven't looked at so far is the conversion between upper and lower case letters. Upper case letters are often referred to as "capitals".

UCASE$

```
Mode(s):  Amiga/Blitz
Function: convert a string of text to upper case
Syntax:   upper$=UCase$(SOURCE$)
```

The UCASE$ function simply converts a source string into upper case characters. Here are some examples:

```
; *** UCase$ example
; *** Filename - UCase$.bb2

; *** Will print "BLITZ BASIC"
Print UCase$("blitz basic")
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

This second example splits up a string of text into individual words and capitalises the first letter of each word. It is a good demonstration of how MID$ can be used to chop up strings:

```
; *** UCase$ example2
; *** Filename - UCase$_Example2.bb2

Print "Please input your full name:"
; *** Input string and convert to lower case
A$=Edit$(80)
A$=LCase$(A$)
A=1
B=1
; *** Chop up string into words
While A<=Len(A$)
  If B=1
    ; *** Convert first letter to capitals
    A$=Mid$(A$,1,A-1)+UCase$(Mid$(A$,A,1))+Mid$(A$,A+1)
    B=0
  End If
  ; *** New word found
  If Mid$(A$,A,1)=" "
    B=1
  End If
  Let A+1
Wend
; *** Output new string
NPrint A$
VWait 50
; *** Return to Blitz Basic 2 editor
End
```

LCASE$

```
Mode(s):  Amiga/Blitz
Function: convert a string of text to lower case
Syntax:   lower$=LCase$(SOURCE$)
```

Working along similar lines, the LCASE$ function converts a source string into lower case characters. For example:

```
; *** Lower and lower
; *** Filename - LCase$.bb2

LOWER$="TOTALLY BLITZED"
LOWER$=LCase$(LOWER$)
; *** Will print "totally blitzed"
Print LOWER$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

You may find it useful, at times, to manipulate numbers in the same way as strings. Unfortunately the Blitz Basic string functions do not support numeric expressions directly. To get around this problem you have to convert numeric variables into text strings using the STR$ function.

STR$

```
Mode(s):  Amiga/Blitz
Function: convert a number into a text string
Syntax:   new$=Str$(EXPRESSION)
```

STR$ converts a numeric variable into a text string. This function allows you to manipulate numbers using string functions. Here is an example:

```
; *** Str$ example
; *** Filename - Str$.bb2

SCORE=100
; *** Convert number to string
S$=Str$(SCORE)
Print S$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

USTR$

```
Mode(s):  Amiga/Blitz
Function: convert a number into a text string
Syntax:   new$=UStr$(EXPRESSION)
```

USTR$ converts a numeric variable into a text string. This function allows you to manipulate numbers using string functions. Unlike STR$, USTR$ is not affected by any active FORMAT commands (consult Chapter 5 for more information on formatting numeric strings). For example:

```
; *** UStr$ example
; *** Filename - UStr$.bb2

A=999
A$=UStr$(A)
; *** Will print "999"
NPrint A$
A$=Left$(A$,1)
; *** Will print "9"
NPrint A$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

The logical opposite of STR$ is the VAL function. VAL converts a numeric string back into a numeric variable.

VAL

```
Mode(s):  Amiga/Blitz
Function: convert a text string into a number
Syntax:   a=Val(SOURCE$)
```

To convert a numeric string into a numeric variable use the VAL function. This conversion will fail, and return (0), if a non-numeric value or second decimal point is found. Here are some examples:

```
; *** Added VALue
; *** Filename - Val.bb2

; *** Will print 180
NPrint Val("180")
; *** Will print 0 (failed conversion)
NPrint Val("ABC")
; *** Will print 0 (failed conversion)
NPrint Val("10.10.10")
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

2.5 Obtaining string information

CHR$

```
Mode(s):  Amiga/Blitz
Function: return a one character string with a given ASCII code
Syntax:   s$=Chr$(CODE)
```

The CHR$ function will return a one-character string equivalent to the specified ASCII code. This little example outputs the whole ASCII character set to the screen:

```
; *** Chr$ example
; *** Filename - Chr$.bb2

For A=32 To 253
  Print Chr$(A)
VWait
Next A
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

ASC

```
Mode(s):  Amiga/Blitz
Function: return the ASCII code of a given character
Syntax:   code=Asc(SOURCE$)
```

ASC will return the ASCII code of the first character in SOURCE$. Try the following examples:

```
; *** Some Asc examples
; *** Filename - Asc.bb2

; *** Returns "32"
NPrint Asc(" ")
; *** Returns "65"
NPrint Asc("A")
; *** Returns "66" ("B")
NPrint Asc("BLITZ BASIC")
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

One of the most important things you need to know about a string is its length. This can be determined as follows.

LEN

```
Mode(s):  Amiga/Blitz
Function: return the length of a string
Syntax:   length=Len(SOURCE$)
```

The LEN function returns the number of characters in SOURCE$. For example:

```
; *** Len example
; *** Filename - Len.bb2

NPrint ""
; *** Input string to test
NPrint "Input your name:"
A$=Edit$(30)
; *** Count characters in string
Print "Your name is ",Len(A$)," letter(s) long"
```

```
VWait 100
; *** Return to Blitz Basic 2 editor
End
```

2.6 Character strings

The following functions are used to convert complex numbers into simple two and four byte text strings in order to save space when writing values to sequential files. Integers, long values, and quick values are currently supported by Blitz Basic (consult Chapter 1 for a full discussion of Blitz Basic types).

2.6.1 Integers

Integers, as we have already established, are whole numbers (e.g. 8, 108, 1008, and 10,008).

MKI$

```
Mode(s):  Amiga/Blitz
Function: return a two byte character string
Syntax:   m$=Mki$(INTEGER)
```

This function creates a two byte character string from the two byte INTEGER parameter. MKI$ is often used when writing integer values to sequential files to save on disk space.

CVI

```
Mode(s):  Amiga/Blitz
Function: logical opposite of MKI$
Syntax:   c=Cvi(STRING$)
```

CVI is the logical opposite of MKI$. The function is used to convert the two byte character string generated by MKI$ back to an integer. Try the following example:

```
; *** Mki$/Cvi example
; *** Filename - Cvi.bb2

If WriteFile (0,"RAM:INTEGER")
  ; *** Open sequential file
  FileOutput 0
  ; *** Convert integer and save to file
  Print Mki$(16705)
  ; *** Close sequential file
  CloseFile 0
  DefaultOutput
  If ReadFile (0,"RAM:INTEGER")
    ; *** Read sequential file
    FileInput 0
    ; *** Convert character string
    NPrint Cvi(Edit$(40))
    ; *** Close sequential file
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

2.6.2 Long values

Long values are integers with much greater range (+/- 2147483648).

MKL$

```
Mode(s):  Amiga/Blitz
Function: return a four byte character string
Syntax:   m$=Mkl$(LONG)
```

The MKL$ function creates a four byte character string from the four byte LONG parameter. MKL$ is often used when writing long values to sequential files to save on disk space.

CVL

```
Mode(s):  Amiga/Blitz
Function: logical opposite of MKL$
Syntax:   c=Cvl(STRING$)
```

CVL is the logical opposite of MKL$. The function is used to convert the four byte character string generated by MKL$ back to a long:

```
; *** Mkl$/Cvl example
; *** Filename - Cvl.bb2

If WriteFile (0,"RAM:LONG")
  ; *** Open sequential file
  FileOutput 0
  ; *** Convert long and save to file
  Print Mkl$($dff000)
  ; *** Close sequential file
  CloseFile 0
  DefaultOutput
  If ReadFile (0,"RAM:LONG")
    ; *** Read sequential file
    FileInput 0
    ; *** Convert character string
    NPrint Hex$(Cvl(Edit$(40)))
    ; *** Close sequential file
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

2.6.3 Quick values

Quick values are fixed-point numeric types, with four decimal point accuracy (e.g. 56.0000, 3.1415 and 45,6789). Quicks are less accurate than floating point numbers but much faster.

MKQ$

```
Mode(s):  Amiga/Blitz
Function: return a four byte character string
Syntax:   m$=Mkq$(QUICK)
```

MKQ$ creates a four byte character string from the four byte QUICK parameter. MKQ$ is often used when writing quick values to sequential files to save on disk space.

CVQ

```
Mode(s):  Amiga/Blitz
Function: logical opposite of MKQ$
Syntax:   c=Cvq(STRING$)
```

CVQ is the logical opposite of MKQ$. The function is used to convert the four byte character string generated by MKQ$ back to a quick. For example:

```
; *** Mkq$/Cvq example
; *** Filename - Cvq.bb2

If WriteFile (0,"RAM:QUICK")
  ; *** Open sequential file
  FileOutput 0
  ; *** Convert quick and save to file
  Print Mkq$(500/7)
  ; *** Close sequential file
  CloseFile 0
  DefaultOutput
  If ReadFile (0,"RAM:QUICK")
```

```
    ; *** Read sequential file
    FileInput 0
    ; *** Convert character string
    NPrint Cvq(Edit$(40))
    ; *** Close sequential file
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

2.7 End-of-Chapter summary

Text strings are surrounded by quotation marks and all string names must end with the dollar ($) character. They can comprise of characters, numbers or even spaces. Strings can be shortened, lengthened, converted to upper and lower case, counted, cloned, searched and even turned into numbers!

Table 2.1 : String functions

Command	Function
ASC	Return ASCII code of character
CASESENSE	Toggle case sensitivity
CENTRE$	Centre a string
CHR$	Return character given ASCII code
CVI	Logical opposite of Mki$
CVL	Logical opposite of Mkl$
CVQ	Logical opposite of Mkq$
INSTR	Search for character in a string
LCASE$	Convert a string into lower case
LEFT$	Return leftmost characters of a string
LEN	Return length of a string
LSET$	Return a string of given length
MID$	Return middle characters of a string
MKI$	Create a two byte string from an integer
MKL$	Create a four byte string from a long
MKQ$	Create a four byte string from a quick
REPLACE$	Replace character in a string
RIGHT$	Return rightmost characters of a string
RSET$	Return a string of given length
STRING$	Clone a string
STRIPLEAD$	Remove leading character
STRIPTRAIL$	Remove ending character
STR$	Convert a number into a string
UCASE$	Convert a string into upper case
UNLEFT$	Remove rightmost characters
UNRIGHT$	Remove leftmost characters

```
USTR$        Convert a number into a string
VAL          Convert a string into a number
```

Chapter 3 : Mathematics

Computers are basically giant number crunchers, so it will come as no great surprise that the Amiga - and Blitz Basic 2 - are great at performing mathematical functions. This chapter will introduce the Blitz Basic maths commands, before moving onto more advanced machine code instructions.

3.1 Arithmetical operators

As explained in Chapter 1, the following operators are used to perform arithmetical operations:

Table 3.1 : Arithmetical operators

```
Operator  Description
========================
*         Multiplication
/         Division
+         Addition
-         Subtraction
^         Exponential
```

For example:

```
; *** Arithmetic
; *** Filename - Arithmetic.bb2

NPrint 2*5 ; *** 2 times 5 = 10
NPrint 20/2 ; *** 20 divided by 2 = 10
NPrint 5+5 ; *** 5 plus 5 = 10
NPrint 15-5 ; *** 15 minus 5 = 10
NPrint 10^1 ; *** 10 to the power 1 = 10
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

The following text provides an insight into Blitz Basic's more powerful mathematical commands.

3.2 Sign on the dotted line

SGN

```
Mode(s):  Amiga/Blitz
Function: return the sign of a number
Syntax:   sign=Sgn(EXPRESSION)
```

The SGN function returns a value which indicates the sign of a number. If the number is negative then (-1) is returned. If the number is zero then (0) is returned, and if the number is positive then (1) is returned.

Table 3.2 : Values returned by SGN

```
Expression  Result
==================
Negative    -1
Zero         0
Positive     1
```

For example:

```
; *** Return the sign of a number
; *** Filename - Sgn.bb2

Print "Input a number:"
; *** Input number to test
E=Edit(40)
; *** Return sign of number
SIGN=Sgn(E)
; *** Number is negative
If SIGN=-1 Then Print "Negative"
; *** Number is equal to 0
If SIGN=0 Then Print "Zero"
; *** Number is positive
If SIGN=1 Then Print "Positive"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ABS

```
Mode(s):  Amiga/Blitz
Function: return an absolute value
Syntax:   a=Abs(EXPRESSION)
```

The ABS function returns the absolute value of a number. It is used to convert a numeric expression into a positive number. This results in the following return values:

Table 3.3 : Values returned by ABS

```
Expression  Abs(EXPRESSION) is:
===============================
Negative    Positive
Zero        0
Positive    Positive
```

For example:

```
; *** Return the absolute of a number
; *** Filename - Abs.bb2

A=Abs(-100)
; *** Returns "100"
Print A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

QABS

```
Mode(s):  Amiga/Blitz
Function: return the absolute value of a quick value
Syntax:   q=QAbs(QUICK)
```

QABS works similarly to ABS except that is uses quick values. Because of this the function operates noticeably quicker than ABS. However, you are limited by the restrictions of the quick type of value. For example:

```
; *** QAbs example
; *** Filename - QAbs.bb2

; *** Returns "16"
Print QAbs(-16)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

3.3 Floating point numbers

FLOATMODE

```
Mode(s):   Amiga/Blitz
Statement: change format of floating point numbers
Syntax:    FloatMode MODE
```

FLOATMODE enables you to change how floating point numbers are output by the PRINT and NPRINT commands. Floating point numbers may be displayed in one of three ways, as follows:

- Exponential format (Mode 1)
- Standard format (Mode -1)
- Guessed format (Mode 0 - Default mode)

Exponential format displays a floating point number as a value multiplied by ten raised to a power. Standard format displays values as they are (What You See Is What You Get). Guessed format forces Blitz to take a stab-in-the-dark as to the most appropriate mode to use. Here are some examples:

```
; *** FloatMode examples
; *** Filename - FloatMode.bb2

A.f=180.57
NPrint A," best guess"
FloatMode 1
NPrint A," exponential"
FloatMode -1
NPrint A," standard"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

INT

```
Mode(s):  Amiga/Blitz
Function: convert floating point number into integer
Syntax:   i=Int(EXPRESSION)
```

The INT function returns the integer part of a floating point number by rounding down it down to the nearest whole number. For example:

```
; *** INTeresting
; *** Filename - Int.bb2

NPrint "Input a number with decimal places"
; *** Input floating point number
A=Edit(40)
; *** Return integer part of number
Print Int(A)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FRAC

```
Mode(s):  Amiga/Blitz
Function: return the fractional part of an expression
Syntax:   f=Frac(EXPRESSION)
```

The FRAC function returns the fractional part of a number (the figures after the decimal point), thus removing the whole number value:

```
; *** Frac example
; *** Filename - Frac.bb2

; *** Returns "1415925"
Print Frac(Pi)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

QFRAC

```
Mode(s):  Amiga/Blitz
Function: return the fractional part of a quick value
Syntax:   q=QFrac(QUICK)
```

QFRAC returns the fractional part of a quick value. It is significantly faster than FRAC, however it can only use quick values. Here is an example:

```
; *** QFrac example
; *** Filename - QFrac.bb2

; *** Returns ".4"
Print QFrac(3.4)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

3.4 Standard mathematical functions

SQR

```
Mode(s):  Amiga/Blitz
Function: calculate square root
Syntax:   square=Sqr(EXPRESSION)
```

This function calculates the square root of a numeric expression. The expression must be a positive number. When the expression is smaller than zero an error is returned. For example:

```
; *** Square eyed
; *** Filename - Sqr.bb2

; *** Returns "3"
NPrint Sqr(9)
; *** Returns "2.828427"
NPrint Sqr(8)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

EXP

```
Mode(s):  Amiga/Blitz
Function: calculate exponential
Syntax:   exponential=Exp(VALUE)
```

The EXP function is used to return the exponential of a specified value. It calculates the xth power to the base of the number (e=2.1782818284):

```
; *** EXPressive example
; *** Filename - Exp.bb2

Print Exp(1)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LOG10

```
Mode(s):  Amiga/Blitz
Function: return logarithm
Syntax:   a=Log10(VALUE)
```

LOG10 returns the base 10 logarithm of a given value. For example:

```
; *** Log10 example
; *** Filename - Log10.bb2

; *** Returns ".9999999"
Print Log10(10)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LOG

```
Mode(s):  Amiga/Blitz
Function: return natural logarithm
Syntax:   a=Log(VALUE)
```

This returns the natural (base e) logarithm of a given value. For example:

```
; *** Log example
; *** Filename - Log.bb2

; *** Returns "2.302584"
Print Log(10)
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

3.5 Trigonometry

PI

```
Mode(s):  Amiga/Blitz
Function: return the Pi constant
Syntax:   p=Pi
```

This function returns the Pi constant (3.14159265359...). This number is the ratio of the circumference of a circle to its diameter, and is often used in trigonometry-based calculations. There is no actual value for Pi as it goes on for an infinite number of decimal places. In Blitz Basic, Pi is accurate to six decimal places. For example:

```
; *** A slice of Pi
; *** Filename - Pi.bb2

; *** Returns "3.141592"
Print Pi
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SIN

```
Mode(s):  Amiga/Blitz
Function: calculate sine of an angle
Syntax:   s=Sin(ANGLE)
```

The SIN function returns the sine of an angle. One very important graph, the sine wave,is used to model many natural phenomena, including sound and light waves. Because the sine function repeats every 360 degrees the graph of (Y=Sin(X)) is periodic.

The following example uses SIN to display a string on a sine wave:

```
; *** Sine text
; *** Filename - Sine_Text.bb2

; *** Sine wave variables
RADIUS=10
OFFSET=0
```

```
YOFFSET=15
; *** Text string to sine
SINE$="Realtime sine wave text in Blitz BASIC!"
; *** Define palette
PalRGB 0,0,0,0,0
PalRGB 0,1,13,13,13
; *** Open screen and grab its BitMap
Screen 0,1,"Sine text"
ScreensBitMap 0,0
; *** Direct PRINT statement to BitMap
BitMapOutput 0
Use Palette 0
While X<=Len(SINE$)
  ; *** Grab characters, one by one
  TXT$=Mid$(SINE$,OFFSET,1)
  ; *** Y co-ordinate on sine wave
  Y=Sin(XY)*RADIUS
  Let XY+0.1
  If XY6
    XY=0
  End If
  ; *** Output characters
  Locate X,Y+YOFFSET
  Print TXT$
  ; *** Next character
  Let X+1
  Let OFFSET+1
Wend
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

COS

```
Mode(s):  Amiga/Blitz
Function: calculate cosine of an angle
Syntax:   c=Cos(ANGLE)
```

The COS function returns the cosine of an angle. The graph of (Y=Cos(X)) is exactly the same shape as the sine curve except that it has been translated 90 degrees to the left. For example:

```
; *** Cosine wave
; *** Filename - Cos.bb2

; *** Define palette
PalRGB 0,0,0,0,0
PalRGB 0,1,13,13,13
; *** Open BLITZ mode display
```

```
BLITZ
BitMap 0,320,DispHeight,1
Slice 0,44,320,DispHeight,$fff8,1,8,2,320,320
Use Palette 0
Show 0
; *** Draw line at centre of cosine wave
Line 0,100,320,100,1
; *** Plot simple cosine wave
For A=1 To 3000
  Plot 10+A/10,Cos(Pi*2*A/1000)*80+100,1
Next
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

TAN

```
Mode(s):  Amiga/Blitz
Function: calculate tangent of an angle
Syntax:   t=Tan(ANGLE)
```

The TAN function returns the tangent of an angle. The following example uses TAN to create tangent contours (a silly idea of mine):

```
; *** Tangent contours
; *** Filename - Tan.bb2

; *** Define palette
PalRGB 0,0,0,0,0
PalRGB 0,2,0,6,0
; *** Open Intuition screen
Screen 0,3,"Hello"
ScreensBitMap 0,0
; *** Grab user palette
Use Palette 0
Cls 2
; *** Plot tangent wave
For B=1 To 4000
  Plot B/10,Tan(Pi+B)*80+80,1
Next
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ASIN

```
Mode(s):  Amiga/Blitz
Function: calculate arc sine
Syntax:   a=ASin(NUMBER)
```

ASIN calculates the angle needed to generate a value with SIN (an arc sine).

ACOS

```
Mode(s):  Amiga/Blitz
Function: calculate arc cosine
Syntax:   a=ACos(NUMBER)
```

Similarly, ACOS returns the arc cosine of a given number.

ATAN

```
Mode(s):  Amiga/Blitz
Function: calculate arc tangent
Syntax:   a=ATan(NUMBER)
```

The ATAN function returns the arc tangent of a given number.

HSIN

```
Mode(s):  Amiga/Blitz
Function: calculate hyperbolic sine
Syntax:   h=HSin(ANGLE)
```

The HSIN function calculates the hyperbolic sine of an angle.

HCOS

```
Mode(s):  Amiga/Blitz
Function: calculate hyperbolic cosine
Syntax:   h=HCos(ANGLE)
```

This function is used to find the hyperbolic cosine of an angle.

HTAN

```
Mode(s):  Amiga/Blitz
Function: calculate hyperbolic tangent
Syntax:   h=HTan(ANGLE)
```

HTAN returns the hyperbolic tangent of an angle.

3.6 Random numbers

Blitz Basic 2 comes complete with an inbuilt function to generate random numbers. Actually the numbers aren't really random as they're the result of a decision made by the computer. If you knew how Blitz created its random numbers then you'd be able to predict exactly which "random number" it would select next.

It'd be pretty difficult to do this though because the computer chooses each number from a very long list, and then repeats the list when it gets to the end. It would be almost impossible to figure out when the list began again.

Public awareness of random numbers and of the laws of probability has increased greatly since the launch of the National Lottery. Whilst Blitz Basic can't predict the results of the Lottery, it can be used to generate a personalised set of numbers for you!

Right, now you've got the basic concept behind random numbers here's how Blitz Basic generates them.

RND

```
Mode(s):  Amiga/Blitz
Function: generate a random number
Syntax:   r=Rnd[(NUMBER)]
```

The RND function returns a random integer between zero and NUMBER. If the optional NUMBER parameter is not included then a random decimal between (0) and (1) is returned. Here is an example:

```
; *** Plot Starfield
; *** Filename - Plot_Example.bb2

; *** Number of stars in starfield
STARS=100
; *** Define palette (lots of whites and greys)
PalRGB 0,0,0,0,0
PalRGB 0,1,10,10,10
PalRGB 0,2,7,7,7
PalRGB 0,3,3,3,3
; *** Pop into Blitz mode
BLITZ
```

```
; *** Open BitMap
BitMap 0,320,DispHeight,2
; *** Plot starfield at random co-ordinates
For A=0 To STARS
  Plot Rnd(320),Rnd(DispHeight),Rnd(3)+1
Next A
; *** Open slice to display BitMap graphics
Slice 0,44,320,DispHeight,$fff8,2,8,8,320,320
; *** Grab user palette
Use Palette 0
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

3.7 Machine code

Instead of using the decimal system, with ten as its base, computers use a form of binary called hexadecimal (or hex for short), based on sixteen. As there are only ten digits available in our number system we need six extra digits to do the counting. So we use A, B, C, D, E and F. And what comes after F? Just as we, with ten fingers, write 10 for ten, so computers write 10 for sixteen.

HEX$

```
Mode(s):  Amiga/Blitz
Function: convert a decimal number into a hexidecimal number
Syntax:   h=Hex$(VALUE)
```

The HEX$ function converts numbers from the decimal system into hexadecimal numbers. The hexadecimal system counts in units of 16 rather than 10, so a total of 16 different digits is needed to represent the different numbers. The digits from 0 to 9 are used as normal, but the digits from 10 to 15 are signified by the letters A to F inclusive.

The decimal value to be converted is specified in brackets. The following chart shows the first 15 hexadecimal numbers, along with their decimal equivalents.

Table 3.4 : Hexadecimal notation

```
Hex digit: 0 1 2 3 4 5 6 7 8 9 A  B  C  D  E  F
Decimal:   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Here are a couple of examples:

```
; *** Hex$ example
; *** Filename - Hex$.bb2
; *** Returns "9"
NPrint Hex$(9)
; *** Returns "A"
NPrint Hex$(10)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

In fact, computers behave as though they had only two digits, represented by a low voltage, or off (0), and a high voltage, or on (1). This is called the binary system, and the two binary digits are called bits: so a bit is either (0) or (1).

BIN$

```
Mode(s):  Amiga/Blitz
Function: convert a decimal number into a binary number
Syntax:   b=Bin$(VALUE)
```

BIN$ converts a decimal number into the equivalent binary number. Here is a short program which prints the first 50 binary and hexadecimal numbers:

```
; *** Bin$ it!
; *** Filename - Bin$.bb2

For A=1 To 50
  NPrint Hex$(A),"  ",Bin$(A)
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

PEEK and POKE are two Basic commands which beginners often have problems with, although they are really very simple. They differ from most Basic in that they act directly on the numbers stored in computer memory. When you PEEK into a memory location the result is the number stored there with POKE.

POKE

```
Mode(s):   Amiga/Blitz
Statement: poke data into a memory location
Syntax:    Poke [.TYPE] ADDRESS,DATA
```

The POKE statement moves a number from zero to 255 into the memory location at the specified address. Here is an example:

```
; *** Poke example
; *** Filename - Poke.bb2

; *** Nip into BLITZ mode
BLITZ
; *** 32 colour display
BitMap 0,320,256,5
Slice 0,44,5
Show 0
X=0
; *** Poke colour register
For A=0 To 5000
  Poke.w $DFF180,X
  Let X+1
  If X=255
    X=0
  End If
Next A
; *** Return to Blitz Basic 2 editor
End
```

PEEK

```
Mode(s):  Amiga/Blitz
Function: return the contents of a memory location
Syntax:   p=Peek [.TYPE](ADDRESS)
```

The PEEK function returns a single 8-bit byte from a memory location at the specified address. For example:

```
; *** Peek example
; *** Filename - Peek.bb2

; *** Put number 39 in address 10
Poke 10,39
; *** Grab contents of address 10
NPrint Peek(10)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

PEEKS$

```
Mode(s):  Amiga/Blitz
Function: return a string of peeked bytes
Syntax:   p$=Peeks$(ADDRESS,LENGTH)
```

PEEK$ reads the maximum number of characters specified in the LENGTH parameter, into a new string. The ADDRESS parameter is the location of the first character to be read:

```
; *** Peek$ example
; *** Filename - Peek$.bb2

; *** Put Blitz Basic in address 10
Poke$ 10,"Blitz Basic"
; *** Grab contents of address 10
NPrint Peek$(10)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CALL

```
Mode(s):   Amiga/Blitz
Statement: call a machine code program
Syntax:    Call ADDRESS
```

The CALL statement is used to run a machine code program straight from the memory location specified by the ADDRESS parameter. For example:

```
; *** Call example
; *** Filename - Call.bb2

a.l=AllocMem_(14,1)

; *** Read machine code into memory
For k=0 To 12 Step 2
  Read w.w
  Poke a+k,w
Next

; *** Call machine code program
For B=1 To 10
  Call a
Next B
```

```
MouseWait
FreeMem_ a,14

; *** A machine code program
Data.w $70ff,$33c0,$00df,$f180,$51c8,$fff8,$4e75
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

3.8 End-of-Chapter summary

Blitz Basic supports standard mathematical functions such as exponentials and logarithms.

The powerful trigonometry functions include sine (SIN), cosine (COS) and tangent (TAN).

Random numbers can be generated using the RND function.

The hexadecimal system counts in units of 16 and the binary system uses zeros (0) and ones (1).

Machine code programs can be run from memory using the CALL statement.

Table 3.5 : Mathematics commands

```
Command     Function
====================================================
ABS         Return an absolute value
ACOS        Calculate arc cosine
ASIN        Calculate arc sine
ATAN        Calculate arc tangent
BIN$        Return binary number
CALL        Call a machine code program
COS         Calculate cosine of an angle
EXP         Calculate exponential
FLOATMODE   Change format of floating point numbers
FRAC        Return fractional part of an expression
HCOS        Calculate hyperbolic cosine
HEX$        Return hexadecimal number
HSIN        Calculate hyperbolic sine
HTAN        Calculate hyperbolic tangent
INT         Return an integer
LOG         Return natural logarithm
LOG10       Return logarithm
PEEK        Return contents of memory location
PEEK$       Return a string of peeked bytes
PI          Return Pi constant
POKE        Poke data into memory location
QABS        Return the absolute value of a quick
QFRAC       Return fractional part of a quick
RND         Generate a random number
SGN         Return the sign of a number
SIN         Calculate sine of an angle
```

```
  SQR         Calculate square root
  TAN         Calculate tangent of an angle
```

Chapter 4 : Control Structures

Control structures are those instructions which allow the Blitz Basic programmer to make decisions. There are five main types of control structure: unconditional jumps, conditional jumps and structured tests, conditional loops, unconditional loops, and controlled loops.

The chapter will also show you how to create Interrupt and error-trapping structures, and procedure definitions.

4.1 Unconditional jumps

Unconditional jumps are those which require no decision-making whatsoever - they simply allow branching from one part of a program to another. If computer programs were executed one line after another, and jumping about to different parts of the program was not possible then code would very quickly become messy and inefficient (see later). Here's how program flow can be transferred from the main program to a sub-program, or sub-routine.

GOTO

```
Mode(s):   Amiga/Blitz
Statement: jump to a specified program label
Syntax:    Goto LABEL
```

With the help of a label, positions within a program can be set to allow branching with the GOTO statement. The program execution is then branched to the position of the label. The label can be made up of letters and numbers, however it must end with a colon. When branching to a label with GOTO, the colon is not included in the name. It is not possible to branch out of or into procedures with the GOTO statement. For example:

```
; *** Goto example
; *** Filename - Goto.bb2

NPrint "Jumping to label in a tick..."
VWait 50
; *** Jump to sub-routine
Goto LABEL
; *** Program flow cannot continue
End

; *** Sub-routine
LABEL:
NPrint "Arrived at label"
VWait 50
; *** Return to Blitz Basic 2 editor
End
```

Most programmers (such as myself) hate GOTOS as they make code messy and unreadable, so do use them sparingly!

Another popular unconditional jump is the GOSUB, which is used to branch program flow from the main program, to a sub-routine. A sub-routine is a sort of mini-program within a program. It carries out a particular task, such as updating the display or printing a message, and you can send the computer to it whenever you want this task carried out. This saves writing out the same program lines each time and makes the program shorter and infinitely more readable.

In Blitz Basic, to tell the computer to branch to a sub-routine, you use the GOSUB statement. Sub-routines can be positioned anywhere in your code and can be called as many or as few times as you like.

GOSUB

```
Mode(s):   Amiga/Blitz
Statement: jump to a sub-routine
Syntax:    Gosub LABEL
```

RETURN

```
Mode(s):   Amiga/Blitz
Statement: return from a sub-routine called by Gosub
Syntax:    Return
```

The GOSUB statement branches program execution to the position of the label (known as a sub-routine). The sub-routine is terminated by the RETURN statement. Unlike GOTO, GOSUB remembers the location of the command immediately after the GOSUB (known as the stack). The RETURN statement branches program execution back to the stack. This method allows one part of a program to be accessed by many other parts of the same program. Here is an example:

```
; *** Gosub and Return example
; *** Filename - Gosub_Return.bb2

For A=1 To 3
  NPrint "This is the main program."
  VWait 50
  ; *** Jump to sub-routine
  Gosub LABEL2
Next A
; *** Return to Blitz Basic 2 editor
End

; *** Sub-routine
LABEL2:
NPrint "This is the sub-routine."
VWait 50
NPrint "(Returning to main program)"
```

```
VWait 50
; *** Return to main program
Return
; *** Program flow cannot continue
End
```

POP

```
Mode(s):   Amiga/Blitz
Statement: exit from a program loop
Syntax:    Pop For/Gosub/If/Repeat/Select/While
```

On occasions it may be necessary to exit from a particular type of program loop in order to branch program execution to a different part of the program. POP is used to exit from jumps (both conditional and unconditional), structured tests and conditional and controlled loops:

Table 4.1 : Control structures which can be POPped

Control structure	Pop?
GOSUB	Y
IF...ENDIF	Y
WHILE...WEND	Y
REPEAT...UNTIL	Y
FOR...TO...NEXT	Y
SELECT...CASE...END SELECT	Y

For example:

```
; *** Pop example
; *** Filename - Pop!.bb2
; *** Call sub-routine forever
Repeat
  Gosub JOYSTICK
Forever

MOOSE:
NPrint "Pop has called the MOUSE sub-routine"
VWait 50
; *** Return to Blitz Basic 2 editor
End

JOYSTICK:
NPrint "Press left mouse button"
; *** Exit sub-routine upon joystick event
If Joyb(0)=1 Then Pop Gosub : Goto MOOSE
Return
```

```
; *** Program flow cannot continue
End
```

4.2 Conditional jumps and structured tests

Often you will want to execute different parts of a program, depending on the outcome of an expression. This is called a conditional jump, or structured test, depending upon the test format.

IF [THEN]

```
Mode(s):   Amiga/Blitz
Statement: choose between alternative actions
Syntax:    If EXPRESSION Then STATEMENTS
```

This command structure makes it possible to make one or more instructions operational only when a logical condition if fulfilled. IF a condition is true THEN the following statements are executed. Here is an example:

```
; *** If...Then example
; *** Conditional test
; *** Filename - Iffy.bb2

NPrint "Input your age (in years):-"
A=Edit(40)
If A<40 Then Print "You are young!"
If A>=40 Then Print "You are over the hill!"
VWait 100
; *** Return to Blitz Basic 2 editor
End
```

AND OR

```
Mode(s):   Amiga/Blitz
Statement: qualify a condition
Syntax:    If CONDITION1 AND CONDITION2 Then STATEMENT
Syntax 2:  If CONDITION1 OR CONDITION2 Then STATEMENT
```

The logical AND operator can also be used to qualify a condition. If CONDITION1 is true, and CONDITION2 is true, then STATEMENT is executed.

The OR operator can be used in the same way. If CONDITION1 or CONDITION 2 is true, then STATEMENT is executed.

Here is a working example:

```
; *** AND...OR example
; *** Filename - AND...OR.bb2

A=5
B=5
C=11
; *** A=B and C is >10
If A=B AND C>10 Then NPrint "True"
; *** A=B, but C is >10
If A<B OR C>10 Then NPrint "Also True"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

NOT

```
Mode(s):   Amiga/Blitz
Statement: negate logical expression
Syntax:    n=NOT EXPRESSION
```

The NOT operator negates a logical expression. It is the only logical operation with one argument. Here are some examples:

```
; *** NOT example
; *** Filename - NOT.bb2

NPrint NOT False ; *** returns -1
NPrint NOT True ; *** returns 0
NPrint NOT 0 ; *** returns -1
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** NOT example 2
; *** Filename - NOT2.bb2

; *** If...Then structure
A=5
B=5
NPrint "A = ",A
NPrint "B = ",B
If A=B Then NPrint "A=B"
```

```
NPrint ""

; *** NOT (negate) structure
A=5
B=10
NPrint "A = ",A
NPrint "B = ",B
If NOT A=B Then NPrint "A<>B"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ENDIF

```
Mode(s):   Amiga/Blitz
Statement: terminate a structured test
Syntax:    If STRUCTURED_TEST
           EndIf
```

However, the IF...THEN conditional jump has been superseded in most BASIC languages by the infinitely more powerful IF...ELSE...ENDIF structured test. If the optional THEN command is omitted then the test becomes a structured one and must be terminated with the ENDIF command. This allows you to execute many lines of Blitz code depending on the outcome of a single condition:

```
; *** If...EndIf example
; *** Structured test
; *** Filename - EndIffy.bb2

NPrint "Input your age (in years):-"
; *** Input some numbers
A=Edit(40)
If A<40
  Print "You are young!"
  ; *** If A>40...
  Else
  Print "You are over the hill!"
EndIf
VWait 100
; *** Return to Blitz Basic 2 editor
End
```

ELSE

```
Mode(s):   Amiga/Blitz
Statement: qualify a condition
Syntax:    If CONDITION Then STATEMENT Else STATEMENT2
Syntax 2:  If CONDITION
           LIST OF STATEMENTS
           Else
           LIST OF STATEMENTS
           EndIf
```

ELSE is used in conjunction with IF and THEN (IF and ENDIF in a structured test) to qualify a condition. The commands between IF and ELSE are executed when the logical condition following IF is true. Then program execution continues with the next command in the program.

If the condition following IF is false then the commands after ELSE are executed instead. Here are some examples:

```
; *** Else example 1
; *** Conditional test
; *** Filename - Else1.bb2

; *** Generate a random integer
A=Int(Rnd(100))
NPrint A," is a random number"
; *** Make a decision
If A<50 Then NPrint A," is less than 50" Else Gosub GREAT
VWait 100
; *** Return to Blitz Basic 2 editor
End

; *** Sub-routine
GREAT:
NPrint A," is greater than 50"
Return
```

```
; *** Else example2
; *** Structured test
; *** Filename - Else2.bb2

A=Int(Rnd(100))
NPrint A," is a random number"
If A<50
  NPrint A," is less than 50"
  ; *** If A>50...
  Else
  Gosub GREAT
```

```
EndIf
VWait 100
; *** Return to Blitz Basic 2 editor
End

; *** Sub-routine
GREAT:
NPrint A," is greater than 50"
Return
```

TRUE

```
Mode(s):   Amiga/Blitz
Statement: return logical true (-1)
Syntax:    t=True
```

TRUE returns the logical true of a constant. This is represented by the number (-1). A value of either true (-1) or false (0) is produced every time a conditional test is executed. Try this example:

```
; *** True example
; *** Filename - True.bb2

Screen 0,3+8,"Mawwage"
Window 0,0,20,640,200,0,"Twoo Wove",1,2
For LOOP=1 To 5
  A=Int(Rnd(3))
  B=Int(Rnd(3))
  NPrint "A = ",A
  NPrint "B = ",B
  C=A<>B
  ; *** Logical true
  If C=True
    NPrint "A<>B"
  ; *** Logical false
  Else
    NPrint "A=B"
  EndIf
  NPrint ""
  VWait 20
Next LOOP
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FALSE

```
Mode(s):   Amiga/Blitz
Statement: return logical false (0)
Syntax:    f=False
```

FALSE returns the logical false of a constant. This is represented by the number (0). A value of either true (-1) or false (0) is produced every time a conditional test is executed. Example:

```
; *** False example
; *** Filename - False.bb2

Screen 0,3+8,"Skween"
Window 0,0,20,640,200,$1000,"Whindoow",1,2
A$=Edit$(10)
B=Len(A$)
; *** Logical false
If B=False
  NPrint "NO TEXT WAS ENTERED!"
; *** Logical true
Else
  NPrint "Length = ",B," characters"
EndIf
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SELECT CASE END SELECT

```
Mode(s):   Amiga/Blitz
Statement: hold the result of an expression
Syntax:    Select EXPRESSION
           Case 1
           ; *** Execute if expression = 1
           Case 2
           ; *** Execute if expression = 2
           End Select
```

SELECT examines and stores the result of the specified expression.

The CASE statement is used following SELECT to execute a section of program code when the expression specified by CASE is equivalent to the expression specified by SELECT.

END SELECT is used to terminate a SELECT...CASE control structure. For example:

```
; *** Select example
; *** Filename - Select.bb2

For A=1 To 10
  N=Int(Rnd(3))+1
  Select N
    ; *** Number one generated
    Case 1
      NPrint "One"
    ; *** Number two generated
    Case 2
     NPrint "Two"
    ; *** Number three generated
    Case 3
      NPrint "Three"
  End Select
  VWait 10
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

DEFAULT

```
Mode(s):   Amiga/Blitz
Statement: execute if CASE not satisfied
Syntax:    Default
```

If none of the CASE statements are satisfied then DEFAULT may be used to cause a section of program code to be executed if none of the CASE statements were satisfied. Try the following example:

```
; *** Default example
; *** Filename - Default.bb2

For A=1 To 25
  N=Int(Rnd(10))+1
  Select N
    ; *** Number one generated
    Case 1
      NPrint "One"
    ; *** Number two generated
    Case 2
      NPrint "Two"
    ; *** Number three generated
    Case 3
      NPrint "Three"
    ; *** Case not satisfied (N>3)
    Default
      NPrint "Number greater than 3!"
  End Select
  VWait 10
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ON

```
Mode(s):   Amiga/Blitz
Statement: jump to a label on the result of an expression
Syntax:    On EXPRESSION Goto LIST OF LABELS
Syntax 2:  On EXPRESSION Gosub LIST OF LABELS
```

ON tells a program to branch, via either a GOTO or a GOSUB, to one of a number of program labels depending upon the result of the expression. The program labels must be separated by commas.

If the expression results in one then the first program label will be branched to. If the expression results in two then the second program label will be branched to and so on.

If the result of the expression is negative, or greater than the number of program labels, then program execution will continue from the command following ON. Try the following example on (no pun intended) for size:

```
; *** On example
; *** Filename - On.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Branch to sub-routines in turn
For LOOP=1 To 3
  On LOOP Gosub STARS,LINES,CIRCLES
VWait 50
Next LOOP
; *** Return to Blitz Basic 2 editor
End

; *** Draw 100 stars
STARS:
For A=1 To 100
  Plot Rnd(320),Rnd(256),Rnd(30)+1
Next A
Return
```

```
; *** Draw 100 lines
LINES:
For B=1 To 100
  Line Rnd(320),Rnd(256),Rnd(320),Rnd(256),Rnd(30)+1
Next B
Return

; *** Draw 100 circles
CIRCLES:
For C=1 To 100
  Circle Rnd(320),Rnd(256),Rnd(10)+1,Rnd(30)+1
Next C
Return
```

4.3 Conditional loops

The conditional loop is one of the most powerful BASIC control structures. It is used to repeat a section of code until the condition of the loop is satisfied.

WHILE WEND

```
Mode(s):   Amiga/Blitz
Statement: repeat loop while condition is true
Syntax:    While CONDITION
           LIST OF STATEMENTS
           Wend
```

The WHILE and WEND instructions are used to create a loop which is to be executed as long as a logical condition is true. When a WHILE statement is encountered, its condition is checked and the loop is only executed if the condition is true. When WEND is reached the program execution jumps back to WHILE and the loop is repeated. Try the following examples:

```
; *** While...Wend example ** Filename - While...Wend.bb2

NPrint "Counting to 100:"
VWait 50
; *** Repeat until A=100
While A<100
  Let A+1
  NPrint A
Wend
NPrint "Finished!"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** While...Wend example 2
; *** Filename - While...Wend2.bb2

BLITZ
Mouse On
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Repeat until mouse button is pressed
While Joyb(0)=0
  Line Rnd(320),Rnd(256),MouseX,MouseY,Rnd(5)+1
Wend
; *** Return to Blitz Basic 2 editor
End
```

REPEAT UNTIL

```
Mode(s):   Amiga/Blitz
Statement: repeat loop until condition is satisfied
Syntax:    Repeat
           LIST OF STATEMENTS
           Until CONDITION
```

The instructions REPEAT and UNTIL are used to create a loop which is to be executed until a logical condition exists. When the REPEAT statement is encountered in a program, the loop is executed. Then the logical condition is checked and if is true then the loop is cancelled and program execution continues after the UNTIL instruction. Both commands should occupy their own lines. Here are some examples:

```
; *** Repeat...Until example
; *** Filename - Repeat...Until.bb2

BitMap 0,320,256,3
BLITZ
Slice 0,44,3
Show 0
; *** Repeat loop until mouse button is pressed
Repeat
  Circlef Rnd(320),Rnd(256),Rnd(10)+5,Rnd(6)+1
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Repeat...Until example 2
; *** Filename - Repeat...Until2.bb2

PalRGB 0,1,15,15,15
BLITZ
Mouse On
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Use Palette 0
; *** Plot a random starfield
For A=1 To 15
  Plot Int(Rnd(320)),Int(Rnd(50)),Rnd(5)+2
Next A
For B=1 To 15
  ; *** Search for coloured stars
  Repeat
    X=Int(Rnd(320))
    Y=Int(Rnd(50))
  Until Joyb(0)>0 OR Point(X,Y)>1
  ; *** Change coloured star to white
  Plot X,Y,1
; *** Loop back until all stars recoloured
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

4.4 Unconditional loops

Unconditional loops, like unconditional jumps, require no decision-making whatsoever. These loops are used to repeat a section of code forever, hence the following Blitz Basic keyword.

FOREVER

```
Mode(s):   Amiga/Blitz
Statement: cause a Repeat loop to repeat infinitely
Syntax:    Repeat
           LIST OF STATEMENTS
           Forever
```

FOREVER is used instead of UNTIL in a REPEAT...UNTIL loop to create an endless loop. The program executes the commands between REPEAT and UNTIL and branches back to REPEAT when UNTIL is reached. Both commands should occupy their own lines, as in the following example:

```
; *** Repeat...Forever example
; *** Filename - Repeat...Forever.bb2

; *** Repeat loop forever
Repeat
  NPrint "This loop will never end. Don't run this example!"
Forever
; *** This command is never reached
End
```

4.5 Controlled loops

Often you need to do the same thing several times in a program. Although you can repeat part of a program using GOTO, a much better way is to repeat the same lines several times using the FOR...TO...NEXT structure. This is known as a controlled loop as the loop is controlled by the programmer, rather than the user.

FOR TO NEXT

```
Mode(s):   Amiga/Blitz
Statement: repeat loop a specific number of times
Syntax:    For INDEX=FIRST_NUMBER To LAST_NUMBER [Step INC]
```

The FOR...TO...NEXT loop repeats a list of instructions a specified number of times. INDEX counts the number of times the loop is repeated and is increased by one each time the loop repeats. The number that INDEX is increased by (or decreased by) can be altered by including the Step INC parameter (see later). At the start of the loop, the INDEX counter is loaded with the FIRST_NUMBER value and is increased each program loop until it reaches the LAST_NUMBER value. Here is an example:

```
; *** For...To...Next example
; *** Filename - For...To...Next.bb2

BitMap 0,320,256,5
BLITZ
Slice 0,44,5
Show 0
BitMapOutput 0
NPrint "200 rectangles!"
VWait 100
Cls
; *** Simple For...To...Next loop (200 loops)
For A=1 To 200
  Boxf Rnd(320),Rnd(256),Rnd(320),Rnd(256),Rnd(30)+1
Next A
VWait 50
Cls
```

```
NPrint "1000 circles!"
VWait 100
Cls
; *** A larger For...To...Next loop (1000 loops)
For B=1 To 1000
  Circlef Rnd(320),Rnd(256),Rnd(10)+5,Rnd(30)+1
Next B
VWait 50
; *** Return to Blitz Basic 2 editor
End
```

If the optional Step INC parameter is included then INC will be added to the counter after each loop instead of one (the default). If the value for INC is negative then the entire loop will be performed in reverse. For example:

```
; *** For..To...Next example 2
; *** Filename - For...To...Next2.bb2

BLITZ
BitMap 0,320,256,1
BitMapOutput 0
Slice 0,44,1
Show 0
Locate 0,0
; *** Step 2 loop
For A=1 To 10 Step 2
  NPrint A
Next A
VWait 50
Locate 0,0
; *** Decreasing loop
For B=10 To 1 Step -1
  NPrint B
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

4.6 Interrupt handling

Interrupts are hardware signals which cause the Amiga's processor to stop what it is doing (usually the execution of the main program) and execute a pre-defined piece of code called an interrupt routine, or interrupt handler. Once the interrupt handler has finished executing, the main program is restarted as if nothing has happened.

There are 14 different types of interrupt on the Amiga:

Table 4.2 : Blitz Basic Interrupts

Interrupt	Cause of Interrupt
0	Serial transmit buffer empty
1	Disk block read/written
2	Software interrupt
3	CIA ports interrupt
4	Copper interrupt
5	Vertical blank
6	Blitter finished
7	Audio channel 0 pointer/length fetched
8	Audio channel 1 pointer/length fetched
9	Audio channel 2 pointer/length fetched
10	Audio channel 3 pointer/length fetched
11	Serial receive buffer full
12	Floppy disk sync
13	External interrupt

Interrupt handlers should never access string variables or literal strings. In Amiga mode no Blitter, Intuition or file access command may be executed by interrupt handlers.

The most useful interrupt is the vertical blank interrupt (number 5). This interrupt occurs every time a vertical blank period has elapsed (about every sixtieth of a second). Consequently, any code defined as a vertical blank interrupt is executed every sixtieth of a second. Vertical blank interrupt handlers must never take longer than one sixtieth of a second to execute, otherwise you are asking for trouble!

SETINT

```
Mode(s):   Amiga/Blitz
Statement: declare code as interrupt code
Syntax:    SetInt TYPE
```

END SETINT

```
Mode(s):   Amiga/Blitz
Statement: end interrupt code
Syntax:    End SetInt
```

The SETINT statement is used to define interrupt code. Any code which appears within an interrupt definition is executed every time the specified interrupt occurs. END SETINT is used to terminate an interrupt definition.

In the first example I am using the vertical blank interrupt to modify a colour register during vertical blank periods, and in the second the same interrupt is used to flash the Amiga's power light:

```
; *** SetInt example 1
; *** Filename - SetInt1.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0

; *** Create background task
SetInt 5
  Let A+1
  Poke.w $dff180,A
End SetInt

; *** Main program
For B=1 To 3000
  Circlef Rnd(320),Rnd(200)+50,Rnd(20)+10,Rnd(5)+1
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** SetInt example 2
; *** Filename - SetInt2.bb2

BLITZ
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
D=1
; *** Create background task
SetInt 5
  ; *** Toggle power light rapidly
  If D=0
    Filter On
  Else
    Filter Off
  EndIf
  D=1-D
End SetInt

; *** Main program
For B=1 To 3000
  Colour Rnd(5)+1
  Locate Rnd(40),Rnd(20)+10
  Print "Hello"
```

```
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CLRINT

```
Mode(s):   Amiga/Blitz
Statement: remove interrupt handler
Syntax:    ClrInt TYPE
```

CLRINT is used to remove an interrupt handler. The TYPE parameter specifies the interrupt type. For example:

```
; *** ClrInt example
; *** Filename - ClrInt.bb2

BLITZ
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
D=1
; *** Create background task
SetInt 5
  Cls Rnd(5)+1
End SetInt
; *** Wait for a mouse click
MouseWait
; *** Remove background task
ClrInt 5
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

4.7 Error handling

Normally any runtime errors which may occur are reported by Blitz Basic's direct mode. However, it is often useful to trap these errors before they are reported by Blitz; this is where custom error handling come in. Custom error handlers are often used during the development stage, and removed once all of the bugs - or "undocumented features", as some programmers refer to them - have been ironed out.

SETERR

```
Mode(s):   Amiga/Blitz
Statement: declare error handler
Syntax:    SetErr
```

END SETERR

```
Mode(s):   Amiga/Blitz
Statement: end error handling definition
Syntax:    End SetErr
```

The SETERR statement defines a custom error handler. Any program code which appears inside a custom error handler will be executed when any Blitz Basic runtime errors occur. Error handlers should be terminated with the END SETERR statement.

This will work fine until the program tries to blit a shape and it is discovered that there are no shape objects in memory:

```
; *** SetErr example 1
; *** Filename - SetErr1.bb2

; *** Create error handler
SetErr
  NPrint "Error!!!"
  NPrint "Press Left Mouse Button"
  ; *** Wait for a mouse click
  MouseWait
  ; *** Return to Blitz Basic 2 editor
  End
End SetErr
; *** There is no object to blit!
Blit 0,100,100
; *** These commands are never reached
MouseWait
End
```

The second error handler generates an error when the program tries to access an array which has too few dimensions:

```
; *** SetErr example 2
; *** Filename - SetErr2.bb2

; *** Create error handler
SetErr
```

```
  NPrint "Error!!!"
  NPrint "Press Left Mouse Button"
  ; *** Wait for a mouse click
  MouseWait
  ; *** Return to Blitz Basic 2 editor
  End
End SetErr
; *** A$ has too few dimensions!
Dim A$(3)
; *** Generate dimension overflow error
For B=1 To 4
  A$(B)="Blitz Basic "+Str$(B)
  NPrint A$(B)
Next B
; *** These commands are never reached
MouseWait
End
```

CLRERR

```
Mode(s):   Amiga/Blitz
Statement: remove error handler
Syntax:    ClrErr
```

This statement is used to remove a custom error handler. For example:

```
; *** ClrErr example ** Filename - ClrErr.bb2
; *** Create custom error handler
SetErr
  NPrint "This error handler will never work"
End SetErr
; *** Remove custom error handler
ClrErr
; *** Generate error
Blit 0,100,100
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ERRFAIL

```
Mode(s):   Amiga/Blitz
Statement: cause a normal error within error handler
Syntax:    ErrFail
```

The ERRFAIL statement is used to temporarily suspend the custom error handler (i.e. the error occurs as normal). Error reporting is then returned to direct mode. Here is an example:

```
; *** ErrFail example
; *** Filename - ErrFail.bb2

; *** Define error handler
SetErr
  NPrint "Error!!!"
  NPrint ""
  NPrint "Press left mouse button to print"
  NPrint "actual error in direct mode."
  ; *** Wait for a mouse click
  MouseWait
  ; *** Suspend error handler
  ErrFail
End SetErr
; *** Create error
Dim A$(3)
For B=1 To 4
  A$(B)="Blitz Basic "+Str$(B)
  NPrint A$(B)
Next B
; *** These commands are never reached
MouseWait
End
```

4.8 Procedures

A procedure is a specially defined module of code that can be called from your main program. Blitz Basic 2 supports two types of procedure, the function-type procedure and the statement-type procedure. A procedure which does not return a value is known as a statement and a procedure which does return a value is known as a function. Both are able to use their own local variables and may gain access to global variables through the use of the SHARED statement.

You may pass up to six variables to a procedure. These variables must be of primitive type. NewType variables may not be used.

4.8.1 Statement-type procedures

STATEMENT

```
Mode(s):   Amiga/Blitz
Statement: create a statement-type procedure
Syntax:    Statement NAME{}
Syntax 2:  Statement NAME{LIST OF OPTIONAL PARAMETERS}
```

END STATEMENT

```
Mode(s):   Amiga/Blitz
Statement: end a statement-type procedure
Syntax:    End Statement
```

A statement-type procedure is created by defining the statement with the STATEMENT statement. If the optional list of parameters are included then parameters may be passed to the procedure. The procedure must be closed with the END STATEMENT statement. Procedures must be called up as follows:

```
NAME{}
```

or:

```
NAME{LIST OF OPTIONAL PARAMETERS}
```

Why not try the following examples:

```
; *** Statement-type procedure
; *** Filename - Statement.bb2

; *** Define procedure
Statement AGE{A}
  NPrint "You are ",A*12," months old."
End Statement

NPrint "Please input your age in years:"
A=Edit(20)
; *** Call procedure
AGE{A}
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Statement-type procedure 2
; *** Filename - Statement2.bb2

; *** Define procedure
Statement UPPER{A$}
  C$=UCASE$(Mid$(A$,1,1))
  B$=Right$(A$,Len(A$)-1)
  D$=C$+B$
  NPrint D$
End Statement

; *** Input name
NPrint "Enter your first name in lower case:-"
A$=Edit$(20)
; *** Call procedure
UPPER{A$}
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

STATEMENT RETURN

```
Mode(s):   Amiga/Blitz
Statement: exit a statement-type procedure immediately
Syntax:    Statement Return
```

STATEMENT RETURN is used to exit from a statement-type procedure before the end of the procedure. Here is an example which exits from the procedure structure once the value (0) is generated:

```
; *** Statement Return
; *** Filename - Statement_Return.bb2

; *** Define procedure
Statement RANDOM{A}
  B=Int(Rnd(A))
  If B=0 Then Statement Return
  NPrint "Random number is: ",B
  ; *** Wait for a mouse click
  MouseWait
  ; *** Return to Blitz Basic 2 editor
  End
End Statement

; *** Input maximum number
NPrint "Input maximum random number:"
A=Edit(20)
; *** Call procedure
RANDOM{A}
; *** These commands are never reached
MouseWait
End
```

4.8.2 Function-type procedures

FUNCTION

```
Mode(s):   Amiga/Blitz
Statement: create a function-type procedure
Syntax:    Function [.TYPE] NAME{}
Syntax 2:  Function [.TYPE] NAME{LIST OF OPTIONAL PARAMETERS}
```

END FUNCTION

```
Mode(s):   Amiga/Blitz
Statement: end a function-type procedure
Syntax:    End Function
```

FUNCTION RETURN

```
Mode(s):   Amiga/Blitz
Statement: exit a function-type procedure immediately
Syntax:    Function Return EXPRESSION
```

The function-type procedure returns a value. It is created by defining the function with the FUNCTION statement. If the optional list of parameters are included then parameters may be passed to the procedure. The procedure must be closed with the END FUNCTION statement.

The optional TYPE parameter may be used to determine what type of result is returned by the function using FUNCTION RETURN. It must be one of Blitz Basic's six primitive variable types:

Table 4.3 : Blitz Basic types

```
Type    Suffix    Example
============================
Byte    .b        Function.b
Word    .w        Function.w
Long    .l        Function.l
Quick   .q        Function.q
Float   .f        Function.f
String  $         Function$
```

If TYPE is omitted then the current default type will be used (default is quick).

FUNCTION RETURN is used within function-type procedures to return the result of the function. Try the following examples:

```
; *** Function-type procedure
; *** Filename - Function.bb2

; *** Define procedure
Function$ HEXBIN{A}
  Function Return Hex$(A)+" "+Bin$(A)
End Function

; *** Call procedure 10 times
For A=1 To 10
  NPrint HEXBIN{A}
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Function-type procedure 2
; *** Filename - Function2.bb2

; *** Define procedure
Function$ BACKWARDS{A$}
  A=Len(A$)
  For LOOP=1 To A
    B$=B$+Mid$(A$,A,1)
    Let A-1
  Next LOOP
  Function Return UCase$(B$)
End Function

; *** Input name
NPrint "Enter your name :-"
A$=Edit$(20)
; *** Call procedure
NPrint BACKWARDS{A$}
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

4.8.3 Global variables

SHARED

```
Mode(s):   Amiga/Blitz
Statement: define a list of global variables
Syntax:    Shared LIST OF VARIABLES
```

Normally, variables inside procedures cannot be written to or read from the main program. We call these variables "local variables" as they can only be used by the procedure itself. All the variables outside of procedures are known as "global variables" - they can be accessed from anywhere.

The SHARED statement takes a list of local variables inside a procedure definition and converts them to global variables, which can be accessed by the main program. This offers an easy way of transferring large amounts of data between procedures. Here is an example:

```
; *** Shared example
; *** Filename - Shared.bb2

SPEED=100
; *** Define procedure
Statement AGE{}
  NPrint SPEED ; *** Prints "0"
  SHARED SPEED
  NPrint SPEED ; *** Prints "100"
End Statement

; *** Call procedure
AGE{}
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

4.8.4 Some useful procedures

Experienced programmers will find that, over the years, they build up a large collection of useful programs, routines and procedures. For the more inexperienced among us, here are three really useful (depending on your point of view) procedures that can be easily incorporated into your own creations, or used independently.

Our first procedure centres a text string on the x-axis. It works by dividing a string in two and positioning one half left of the centre of the display, and the other half right of the centre. At present the procedure only works on low-resolution BitMaps. Try improving it so that it automatically centres a string on any resolution screen or BitMap:

```
; *** Useful Procedure 1
; *** Filename - Proc1.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
BitMapOutput 0

; *** Define procedure
Statement CENTRE{A$}
  ; *** Half screen width and text
  X=40/2-Len(A$)/2
  Locate X,10
  Colour 4
  Print A$
End Statement

; *** Call procedure
A$="Blitz Basic centred text"
CENTRE{A$}
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Our second procedure generates a specified number of random numbers. "But what's wrong with the RND function", I hear you cry! Well, if you were to generate a series of random numbers using RND then the chances are that the same number will come up more than once. This routine can be used to generate any amount of random numbers, and what's more, they will never repeat:

```
; *** Useful Procedure 2
; *** Filename - Proc2.bb2

; *** Number of random numbers
MAX=10
; *** Array to hold numbers
Dim RANDOM(MAX)

; *** Define procedure
Statement RANDOM{MAX}
  SHARED RANDOM()
  ; *** Generate random numbers
  For A=1 To MAX
    RANDOM(A)=A
  Next A
  ; *** Mix them up
  Repeat
```

```
    Repeat
      B=Rnd(MAX)+1
    Until B>0
    Exchange RANDOM(B),RANDOM(MAX)
    Let C+1
  Until C=MAX
End Statement

; *** Call procedure
RANDOM{MAX}
For T=1 To MAX
  NPrint RANDOM(T)
Next T
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Our third and final procedure is used to solve general quadratic equations. It is primarily of use to A-Level Mathematics students (who will understand what a general quadratic equation is!). Simply enter the appropriate values for A, B and C and the procedure will do the rest. Try adding an error generator for quadratic equations which cannot be solved:

```
; *** Useful Procedure 3
; *** Filename - Proc3.bb2

Statement QUADRATIC{A,B,C}
  ; *** Negate B
  B=NOT B
  ; *** Maths jiggery pokery
  D=Sqr((B*B)-(4*A*C))
  E=(B+D)/(2*A)
  F=(B-D)/(2*A)
  ; *** Output answer
  NPrint "X  = ",E
  NPrint "or = ",F
End Statement

; *** Input appropriate values
NPrint "Enter A:"
A=Edit(2)
NPrint "Enter B:"
B=Edit(2)
NPrint "Enter C:"
C=Edit(2)
NPrint ""
; *** Call procedure
QUADRATIC{A,B,C}
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

4.9 End-of-Chapter summary

There are five control structures in Blitz Basic: unconditional jumps (GOTO and GOSUB), conditional jumps and structured tests (IF...ENDIF & SELECT...CASE...END SELECT), conditional loops (WHILE...WEND & REPEAT...UNTIL), unconditional loops (REPEAT...FOREVER), and controlled loops (FOR...TO...NEXT).

Unconditional jumps are those which require no decision-making whatsoever - they simply allow branching from one part of a program to another.

Conditional jumps and structured tests are used to execute different parts of a program, depending on the outcome of an expression.

The conditional loop is used to repeat a section of code until the condition of the loop is satisfied.

Unconditional loops, like unconditional jumps, require no decision-making whatsoever. These loops are used to repeat a section of code forever.

Controlled loops are used to execute the same program lines several times in a program.

Interrupt handlers are defined using SETINT and END SETINT. Interrupts are hardware signals which cause the Amiga's processor to stop what it is doing (usually the execution of the main program) and execute a pre-defined piece of code called an interrupt handler. Vertical blank interrupts are executed every sixtieth of a second.

Runtime errors can be trapped using custom error handlers. These are created with the SETERR and END SETERR statements. Error handlers are suspended with ERRFAIL.

Blitz Basic 2 supports two types of procedure: functions and statements. Procedures which do not return values are known as statements. Procedures which do return values are known as functions. Up to six values can be passed to Blitz Basic 2 procedures.

GOTOs and GOSUBS from inside procedures to labels outside of procedure definitions are illegal.

Variables used in procedure definitions are initialised with every call of the procedure.

Function-type procedures can return any primitive type using the FUNCTION RETURN statement.

Local variables are those contained within procedure definitions and may only be used by procedures.

Procedures may gain access to global variables through the use of the SHARED statement.

Chapter 5 : Input/Output

This chapter will show you how to output text onto a screen or BitMap. It will aid you in the reading of the keyboard and the joystick and mice ports, and teach you the basics of file access.

5.1 Text

Virtually all computer programs use text. Text can be used to prompt the user for input, or to display a congratulatory message or high-score table. Blitz Basic can be used to create anything from simple messages to fully-blown text adventures!

5.1.1 Printing on screen

PRINT

```
Mode(s):   Amiga/Blitz
Statement: print items on screen
Syntax:    Print EXPRESSION
```

The PRINT statement is used to output numeric variables and strings to the current output channel. PRINT followed by a string variable or expression displays the string or strings they represent (strings must be enclosed in quotation marks). Followed by a numeric expression, PRINT displays the expression's value. Followed by a null string (""), PRINT displays a blank line. Here are some examples which all produce the same output:

```
; *** Print examples
; *** Filename - Print.bb2

Print "Blitz Basic is the best!"
Print "Blitz Basic"+" is the best!"
Print "Blitz Basic"," is the best!"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

NPRINT

```
Mode(s):   Amiga/Blitz
Statement: print items on screen
Syntax:    NPrint EXPRESSION
```

NPRINT is used to output numeric variables and strings to the current output channel. NPRINT followed by a string variable or expression displays the string or strings they represent (strings must be enclosed in quotation marks). Followed by a numeric expression, NPRINT displays the expression's value. Followed by a null string (""), NPRINT displays a blank line.

Unlike PRINT, NPRINT automatically outputs a newline character. For example:

```
; *** NPrint example
; *** Filename - NPrint.bb2

Print "Going "
Print "down"
NPrint ""
NPrint ""
NPrint "Going"
NPrint "Down"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

BITMAPOUTPUT

```
Mode(s):   Amiga/Blitz
Statement: direct PRINT commands to a BitMap
Syntax:    BitMapOutput BITMAP#
```

The BITMAPOUTPUT statement is used to direct all future PRINT or NPRINT statements to a BitMap. Fonts used for BitMap output must be eight-by-eight non-proportional fonts (see later). Here is an example:

```
; *** BitMapOutput example
; *** Filename - BitMapOutput.bb2

; *** Pop into Blitz mode
BLITZ
; *** Open BitMap to display graphics
BitMap 0,320,256,3
; *** Direct PRINT statements to BitMap
BitMapOutput 0
; *** Open a Slice and display BitMap
Slice 0,44,3
Show 0
; *** Output some text
For A=1 To 50
  Locate Rnd(40),Rnd(25)
  Colour Rnd(5)+1
  Print "BitMap Output"
Next A
; *** Wait for a mouse click
```

```
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

DEFAULTOUTPUT

```
Mode(s):   Amiga/Blitz
Statement: send PRINT statements to the default CLI window
Syntax:    DefaultOutput
```

This statement causes all future PRINT and NPRINT statements to send their output to the default CLI window. This is the CLI window the program was run from. For example:

```
; *** DefaultOutput example
; *** Filename - DefaultOutput.bb2

; *** Open an Intuition screen and window
Screen 0,0,100,320,200,3,0,"A Screen",1,2
Window 0,10,40,200,50,0,"A Window",1,2
; *** Output text to window
Print "hello from window"
; *** Direct PRINT statement to CLI window
DefaultOutput
Print "Hello from CLI window"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

5.1.2 Formating numeric strings

FORMAT

```
Mode(s):   Amiga/Blitz
Statement: control output of numeric values
Syntax:    Format STRING$
```

FORMAT is used to control the output of numeric values with PRINT and NPRINT. STRING$ is a string expression, of up to 80 characters in length, containing formatting information:

Table 5.1 : Text formatting

```
Character  Description
====================================================
0          Replace missing digits with zeros
.          Insert decimal point
,          Insert commas every 3 digits to the left
+          Insert sign of value
-          Insert sign of value, if negative
#          Replace missing digits with spaces
```

Here is an example:

```
; *** Amiga Format
; *** Filename - Format.bb2

Format "###.00"
NPrint 156 ; *** Returns "156.00"
Format "+#"
NPrint 5 ; *** Returns "+5"
Format "###,###,###"
NPrint 390000000 ; *** Returns "390,000,000"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

5.1.3 Changing the text style

LOADBLITZFONT

```
Mode(s):   Amiga
Statement: load a new font for BitMap output
Syntax:    LoadBlitzFont FONT#,"FILENAME.FONT"
```

The LOADBLITZFONT statement creates a blitzfont object. Blitzfonts are used in the rendering of text to BitMaps only. The default font is the ROM-resident topaz font, however this can be replaced with the blitzfont of your choice. The "FILENAME.FONT" parameter specifies the name of the font to load, which must be located in the fonts directory of the disk.

LOADBLITZFONT can only be used with eight-by-eight non-proportional fonts. Here is an example:

```
; *** LoadBlitzFont example
; *** Filename - LoadBlitzFont.bb2

; *** Load a BlitzFont into memory
LoadBlitzFont 0,"FILENAME.FONT"
; *** Wait for disk access to finish
VWait 20
BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Direct NPRINT statement to BitMap
BitMapOutput 0
Locate 0,5
Colour 4
NPrint "THE QUICK BROWN FOX JUMPED ETC."
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

USEBLITZFONT

```
Mode(s):   Amiga/Blitz
Statement: select current font
Syntax:    UseBlitzFont FONT#
```

If there is more than one blitzfont in memory then USEBLITZFONT provides an easy method for switching between them. FONT# is the number of the blitzfont to use. For example:

```
; *** Use BlitzFont example
; *** Filename - Use BlitzFont.bb2

; *** Load two BlitzFonts into memory
LoadBlitzFont 0,"FILENAME.FONT"
LoadBlitzFont 1,"FILENAME2.FONT"
; *** Wait for disk access to finish
VWait 20
BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
BitMapOutput 0
Locate 0,5
Colour 4
; *** First BlitzFont
Use BlitzFont 0
```

```
NPrint "THE QUICK BROWN FOX JUMPED ETC."
Colour 2
; *** Second BlitzFont
Use BlitzFont 1
NPrint "OVER THE LAZY DOG..."
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FREEBLITZFONT

```
Mode(s):   Amiga/Blitz
Statement: erase a font from memory
Syntax:    FreeBlitzFont FONT#
```

FREEBLITZFONT erases a specified blitzfont from memory. This frees any memory previously occupied by the font. Here's an example:

```
; *** FreeBlitzFont example
; *** Filename - FreeBlitzFont.bb2

; *** Load a BlitzFont into memory
LoadBlitzFont 0,"FILENAME.FONT"
; *** Wait for disk access to finish
VWait 20
BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
BitMapOutput 0
NPrint "THE QUICK BROWN FOX JUMPED ETC."
; *** Remove BlitzFont from memory
FreeBlitzFont 0
NPrint "THE SLOW RED SLUG DUCKED ETC."
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

5.1.4 Setting the text colour

Even if the output of your programs consists just of PRINT or NPRINT statements, there is no reason why they cannot be arranged on the screen in an interesting, clear and attractive way. Use of colour within PRINT statements from time to time can enhance the message and improve its legibility. Random colour changes, for example, can be extremely effective.

COLOUR

```
Mode(s):   Amiga/Blitz
Statement: set the colour of text
Syntax:    Colour FOREGROUND[,BACKGROUND]
```

The COLOUR statement is used to set the colour used to render text to BitMaps. FOREGROUND is the colour of the text and BACKGROUND is the colour of the text background. Here are a couple of examples which illustrate the use of COLOUR in both Blitz and Amiga mode:

```
; *** Colour example 1
; *** Filename - Colour1.bb2

BLITZ
; *** Open a Blitz mode display
BitMap 0,320,256,5
Slice 0,44,5
Show 0
; *** Direct NPRINT statements to BitMap
BitMapOutput 0
For A=1 To 30
  ; *** Select a random colour
  Colour Rnd(30)+1
  NPrint "All the colours of the rainbow(and some)"
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Colour example 2
; *** Filename - Colour2.bb2

; *** Open an Intuition display
Screen 0,5,"My screen"
ScreensBitMap 0,0
; *** Direct NPRINT statements to BitMap
BitMapOutput 0
Locate 0,4
For B=1 To 20
  ; *** Select a random colour
  Colour Rnd(30)+1
  NPrint "A million colours on an Intuition screen"
Next B
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

5.1.5 The text cursor

The position of print output on the screen can also be important, and the LOCATE statement makes it easy to place your information wherever you want it.

LOCATE

```
Mode(s):   Amiga/Blitz
Statement: position the text cursor
Syntax:    Locate X,Y
```

LOCATE positions the text cursor on the current BitMap. The X parameter specifies the horizontal position (rounded down to a multiple of eight) and the Y parameter specifies the vertical position (not rounded). LOCATE must follow a BITMAPOUTPUT statement. For example:

```
; *** A nice location
; *** Filename - Locate.bb2

BLITZ
BitMap 0,640,256,3
Slice 0,44,3+8
Show 0
BitMapOutput 0
Locate 0,0
Colour Int(Rnd(6)+1)
NPrint "Top left"
Locate 36,12
Colour 5
NPrint "Middle"
Locate 67,30
Colour Int(Rnd(6)+1)
NPrint "Bottom right"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CURSX

```
Mode(s):  Amiga/Blitz
Function: return the horizontal position of the text cursor
Syntax:   x=CursX
```

The CURSX statement returns the current horizontal character position of the text cursor. CURSX must follow a BITMAPOUTPUT statement. For example:

```
; *** CursX example
; *** Filename - CursX.bb2

BLITZ
; *** Open a Blitz mode display
BitMap 0,640,256,3
Slice 0,44,3+8
Show 0
BitMapOutput 0
For A=1 To 5
  ; *** Randomly locate cursor
  Locate Int(Rnd(60)),Int(Rnd(30))
  Colour 2
  Print ""
  ; *** Return cursor location
  X=CursX
  Locate 0,0
  Colour 4
  NPrint "X = ",X
  VWait 100
  Cls
Next A
; *** Return to Blitz Basic 2 editor
End
```

CURSY

```
Mode(s):  Amiga/Blitz
Function: return the vertical position of the text cursor
Syntax:   y=CursY
```

The CURSY statement returns the current vertical character position of the text cursor. CURSY must follow a BITMAPOUTPUT statement. Here is an example:

```
; *** CursX/Y example
; *** Filename - CursY.bb2

BLITZ
; *** Open a Blitz mode display
BitMap 0,640,256,3
Slice 0,44,3+8
Show 0
BitMapOutput 0
For A=1 To 5
```

```
    ; *** Randomly locate cursor
    Locate Int(Rnd(60)),Int(Rnd(30))
    Colour 2
    Print "Hello"
    ; *** Return cursor location
    X=CursX
    Y=CursY
    Locate 0,0
    Colour 4
    NPrint "Hello X = ",X
    NPrint "Hello Y = ",Y
    VWait 100
    Cls
  Next A
  ; *** Return to Blitz Basic 2 editor
  End
```

CURSOR

```
Mode(s):   Amiga
Statement: set the thickness of the text cursor
Syntax:    Cursor THICKNESS
```

The CURSOR statement is used to set the thickness of the text cursor. If THICKNESS is negative then a block cursor will be used, otherwise an underline cursor, THICKNESS pixels high will be used. Try the following example:

```
; *** Cursor thickness
; *** Filename - Cursor.bb2

Screen 0,3+8,"My screen"
Window 0,0,20,320,200,$1000,"Cursors",0,1
NPrint "This is a block cursor:"
A$=Edit$(10)
Cursor 1
NPrint "This is an underlined one:"
A$=Edit$(10)
; *** Return to Blitz Basic 2 editor
End
```

5.2 The Keyboard

Input is a simple term which means the feeding of information into the computer where it is processed. Such processing may be addition and subtraction of numbers or storing information such as text. On the Amiga information can be entered using the keyboard. This section covers the commands which can be used to read the Amiga's 96 (I counted every one of them!) key keyboard.

5.2.1 Reading the keyboard

BLITZKEYS

```
Mode(s):   Blitz
Statement: toggle Blitz mode keyboard reading
Syntax:    BlitzKeys On
```

The BLITZKEYS statement is used to toggle Blitz mode keyboard reading (note that BlitzKeys Off is no longer supported by Blitz Basic 2). If keyboard reading is enabled then the keyboard can be read in Blitz mode. For example:

```
; *** BlitzKeys example
; *** Filename - BlitzKeys.bb2

BLITZ
; *** Open a Blitz mode display
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
; *** Enable Blitz mode keyboard reading
BlitzKeys On
NPrint "Type some rubbish..."
; *** Input some text
While Joyb(0)=0
  Print Inkey$
Wend
; *** Return to Blitz Basic 2 editor
End
```

Note that the following commands can only work in Blitz mode if Blitz mode keyboard reading is enabled, as in the above example.

BLITZREPEAT

```
Mode(s):   Blitz
Statement: vary Blitz mode key repeat delays
Syntax:    BlitzRepeat DELAY,SPEED
```

The BLITZREPEAT statement is no longer supported by Blitz Basic 2. As such there is no example.

INKEY$

```
Mode(s):  Amiga/Blitz
Function: check for a key-press
Syntax:   i$=Inkey$[(CHARACTERS)]
```

This function is used to detect the pressing of keys on the keyboard. INKEY$ requires no arguement and is generally used to assign a character to a string variable or to test for a particular character. If no key is being pressed, then INKEY$ returns a null string (""). Note that INKEY$ distinquishes between capital and lower-case letters. If the optional CHARACTERS parameter is included then more than one character (the default) may be collected. For example:

```
; *** Inkey$ example
; *** Filename - Inkey$.bb2

Screen 0,3+8,"Screen"
; *** Open window to output text
Window 0,0,20,320,200,$1000,"My word, another window",1,2
NPrint "Type some rubbish..."
Repeat
  ; *** Wait for a key-press
  WaitEvent
  Print Inkey$
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

RAWSTATUS

```
Mode(s):  Blitz
Function: test for a specific key-press
Syntax:   k=RawStatus(RAW_CODE)
```

Use this function to determine whether or not a specific key is being pressed. RAW_CODE is the raw code of the key to be tested. If the key is being pressed then a value of (-1) will be returned, otherwise (0) will be returned. For example:

```
; *** RawStatus example
; *** Filename - RawStatus.bb2

BLITZ
; *** Open a Blitz mode display
BitMap 0,320,256,3
BitMapOutput 0
```

```
Slice 0,44,3
Show 0
; *** Enable Blitz mode keyboard reading
BlitzKeys On
While Joyb(0)=0
  Locate 0,1
  Print "Del key is : "
  ; *** Test status of DEL key
  If RawStatus(70)
    Print "down"
  Else
    Print "up   "
  EndIf
Wend
; *** Return to Blitz Basic 2 editor
End
```

EDIT$

```
Mode(s):  Amiga/Blitz
Function: input a text string
Syntax:   e$=Edit$([DEFAULT,]CHARACTERS)
```

EDIT$ enables text strings to be entered during the execution of a program, with or without an input cursor (DEFAULT). The cursor is always positioned at the last cursor position. CHARACTERS specifies the number of characters that can be inputed with EDIT$. For example:

```
If the EDIT$ function follows a WINDOWINPUT command then EDIT$ will input
from and output to the current window, whereas a preceding FILEINPUT
command will cause EDIT$ to receive its input from a file.
```

For example:

```
; *** Edit$ example 1
; *** Filename - Edit$1.bb2

; *** Open an Intuition display
Screen 0,3+8,"A Screen"
Window 0,0,20,200,200,$1000,"Window",1,0
NPrint "Enter your first name..."
; *** Input some text (10 characters max)
A$=Edit$(10)
NPrint "Hello ",A$
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Edit$ example 2
; *** Filename - Edit$2.bb2

; *** Open an Intuition display
Screen 0,3+8,"Another Screen"
Window 0,0,20,200,200,$1000,"Another Window",1,0
NPrint "Enter your first name..."
; *** Input some text (with default prompt)
A$=Edit$("Default name",12)
NPrint "Hello ",A$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

EDIT

```
Mode(s):  Amiga/Blitz
Function: input a numeric value
Syntax:   e=Edit([DEFAULT,]CHARACTERS)
```

EDIT enables numbers to be entered during the execution of a program, with or without an input cursor (DEFAULT). The cursor is always positioned at the last cursor position. CHARACTERS specifies the number of characters that can be inputed with EDIT. For example:

```
If the EDIT function follows a WINDOWINPUT command then EDIT will input
from and output to the current window, whereas a preceding FILEINPUT
command will cause EDIT to receive its input from a file.
```

For example:

```
; *** Edit example
; *** Filename - Edit.bb2

NPrint "Enter a number:"
; *** Input a number (10 characters max)
A=Edit(10)
NPrint "Your number was ",A
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

DEFAULTINPUT

```
Mode(s):   Amiga/Blitz
Statement: receive Edit$ input from CLI window
Syntax:    DefaultInput
```

DEFAULTINPUT forces all future EDIT$ functions to receive input from the CLI window the program was run from. This is the default channel used when a Blitz Basic program is first run. For example:

```
; *** DefaultInput example
; *** Filename - DefaultInput.bb2

; *** Open an Intuition display
Screen 0,0,100,320,200,3,0,"A Screen",1,2
Window 0,10,40,200,50,$1000,"A Window",1,2
NPrint "Enter text into window"
; *** Input some text (10 characters max)
A$=Edit$(10)
; *** Send input/output to CLI window
DefaultInput
DefaultOutput
; *** Remove window from display
CloseWindow 0
NPrint "Enter some text into CLI window"
; *** Input some more text (10 characters max)
B$=Edit$(10)
; *** Return to Blitz Basic 2 editor
End
```

BITMAPINPUT

```
Mode(s):   Blitz
Statement: enable Edit & Edit$ in Blitz mode
Syntax:    BitMapInput
```

The BIMAPINPUT statement enables the EDIT and EDIT$ functions in Blitz mode. A BLITZKEYS ON statement must have been executed prior to BITMAPINPUT, otherwise it will not function correctly. A BITMAPOUTPUT statement must also be executed before an EDIT or EDIT$ function. Here is an example:

```
; *** Using Edit$ in Blitz mode
; *** Filename - BitMapInput.bb2

BLITZ
; *** Open a Blitz mode display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Direct text output to BitMap
BitMapOutput 0
; *** Enable Blitz mode keyboard reading
BlitzKeys On
BitMapInput
Locate 0,2
; *** Input some text (9 characters max)
A$=Edit$("Type away",9)
; *** Return to Blitz Basic 2 editor
End
```

5.3 The Joystick

A joystick is, if you play games, the single most important peripheral for your Amiga. That little black box of electrical trickery can be used to blast baddies, dodge dinosaurs and drive Diablos! Thankfully Blitz Basic provides us with a number of exciting commands which take full control over the common or garden joystick.

JOYR

```
Mode(s):  Amiga/Blitz
Function: return the status of a joystick
Syntax:   direction=Joyr(PORT)
```

The JOYR function is used to find out in which way the joystick is being waggled. If you want to take a look at the joystick port then you must tell Blitz to investigate port (1). Or, if you want to snoop around the mouse port (if another joystick is connected) then point Blitz towards port (0). Try the following example which prints the status of a joystick in port 1:

```
; *** Joyr example
; *** Filename - Joyr.bb2

; *** Repeat until fire button is pressed
Repeat
  ; *** Output joystick status
  NPrint Joyr(1)
Until Joyb(1)<>0
```

```
; *** Return to Blitz Basic 2 editor
End
```

When you run the program, irrelevant numbers appear on the screen that change according to joystick movement. Wouldn't it be nice if you knew what these numbers meant? Then why not take a look at the table below (for your convenience I have also included corresponding compass bearings).

Table 5.2 : Reading the joystick port using JOYR

```
Bit number  Joystick direction
==============================
0           Up            [N]
1           Up & Right    [NE]
2           Right         [E]
3           Down & Right  [SE]
4           Down          [S]
5           Down & Left   [SW]
6           Left          [W]
7           Up & Left     [NW]
8           No direction
```

JOYX

```
Mode(s):  Amiga/Blitz
Function: return the left/right status of a joystick
Syntax:   direction=Joyx(PORT)
```

This returns a value of (-1) if the joystick connected to the given port number has been moved to the left. If the joystick is held to the right then this value is (1), otherwise a value of (0) is returned (meaning the joystick is held neither left nor right). Here is an example:

```
; *** Joyx example
; *** Filename - Joyx.bb2

BLITZ
BitMap 0,320,256,4
; *** Create BOB
Boxf 1,1,10,10,1
GetaShape 0,0,0,11,11
Cls
Slice 0,44,4
Show 0
; *** Starting co-ordinates of BOB
X=150
Y=100
; *** BitMap storage buffer
Buffer 0,16384
```

```
Repeat
  VWait
  UnBuffer 0
  ; *** Test joystick and move BOB
  If Joyx(1)=-1 AND X>0 Then X-2
  If Joyx(1)=1 AND X<300 Then X+2
  BBlit 0,0,X,Y
Until Joyb(1)>0
; *** Return to Blitz Basic 2 editor
End
```

JOYY

```
Mode(s):  Amiga/Blitz
Function: return the up/down status of a joystick
Syntax:   direction=Joyy(PORT)
```

JOYY works in a similar way to JOYX. It returns a value of (-1) if the joystick connected to the given port is held upwards, and a value of (1) if it is held downwards. Otherwise it returns a value of (0) (meaning the joystick is held neither upwards nor downwards). For example:

```
; *** Joystick control
; *** Filename - Joyy.bb2

BLITZ
BitMap 0,320,256,4
; *** Create BOB
Boxf 1,1,10,10,1
GetaShape 0,0,0,11,11
Cls
Slice 0,44,4
Show 0
; *** Starting co-ordinates of BOB
X=150
Y=100
; *** BitMap storage buffer
Buffer 0,16384
Repeat
  VWait
  UnBuffer 0
  ; *** Test joystick and move BOB
  If Joyx(1)=-1 AND X>0 Then X-2
  If Joyx(1)=1 AND X<300 Then X+2
  If Joyy(1)=-1 AND Y>10 Then Y-2
  If Joyy(1)=1 AND Y<200 Then Y+2
  BBlit 0,0,X,Y
Until Joyb(1)>0
```

```
; *** Return to Blitz Basic 2 editor
End
```

JOYB

```
Mode(s):  Amiga/Blitz
Function: return the button status of the joystick/mouse
Syntax:   button=Joyb(PORT)
```

In order to read the status of either the joystick or mouse buttons you must use the JOYB command, followed by the port number. A value of (1) will be returned only if the left button is held down. If the right button is held down then a value of (2) is returned. You may also find it useful, on some occasions, to test if both buttons are pressed (a value of (3) is returned). Finally, if no buttons are held down then JOYB will graciously return (0). Try the following example:

```
; *** Joyb example
; *** Filename - Joyb.bb2

OK=1
Repeat
  VWait
  A=Joyb(0)
  If A>0
    If A=1 Then NPrint "Left mouse button"
    If A=2 Then NPrint "Right mouse button"
    If A=3
      NPrint "Both buttons"
      VWait 50
      OK=0
    EndIf
    Repeat : Until Joyb(0)=0
  EndIf
Until OK=0
; *** Return to Blitz Basic 2 editor
End
```

5.4 Reading the mouse status

Whereas the joystick has come to be regarded as the tool of the games player, the mouse has had a much wider use. It has been used as a control method for games (as in Populous and Syndicate), and more often for controlling applications (such as those involving Intuition). Blitz Basic's powerful mouse commands can be used to create both.

MOUSE

```
Mode(s):   Amiga
Statement: turn Blitz mode mouse reading on or off
Syntax:    Mouse On/Off
```

The MOUSE statement toggles Blitz mode mouse reading. In order for the following functions to work in Blitz mode, mouse reading must have been previously enabled:

```
; *** Mouse example
; *** Filename - Mouse.bb2

BLITZ
BitMap 0,320,256,2
BitMapOutput 0
Slice 0,44,2
Show 0
Boxf 0,0,10,10,1
GetaShape 0,0,0,10,10
GetaSprite 0,0
Mouse On
Pointer 0,0
While Joyb(0)=0
  VWait
  Locate 0,0
  NPrint "X = ",MouseX,"  "
  NPrint "Y = ",MouseY,"  "
Wend
; *** Return to Blitz Basic 2 editor
End
```

MOUSEX

```
Mode(s):  Blitz
Function: return the current horizontal location of the mouse pointer
Syntax:   x=MouseX
```

MOUSEX is a Blitz mode command whose purpose is to find the current horizontal location of the mouse pointer. Blitz mode mouse reading must have been previously enabled using MOUSE ON. For example:

```
; *** Mouse coordinates
; *** Filename - MouseX.bb2

BLITZ
BitMap 0,320,256,1
Slice 0,44,1
Show 0
BitMapOutput 0
Mouse On
MouseArea 0,0,320,256
Repeat
  Locate 0,0 : Print "X coord: ",MouseX,"  "
  Locate 0,1 : Print "Y coord: ",MouseY,"  "
  VWait
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

MOUSEY

```
Mode(s):  Blitz
Function: return the current vertical location of the mouse pointer
Syntax:   y=MouseY
```

MOUSEY is the vertical equivalent of MOUSEX in that it returns the current vertical location of the mouse pointer. Blitz mode mouse reading must have been previously enabled using MOUSE ON. See above example.

MOUSEXSPEED

```
Mode(s):  Blitz
Function: return the current horizontal direction of mouse movement
Syntax:   xdirection=MouseXSpeed
```

MOUSEXSPEED is one of those blindingly obvious commands, whose function is to find the current horizontal speed of mouse movement. Again, Blitz mode mouse reading must have been previously enabled using MOUSE ON. If a negative value is returned, then the mouse has been moved leftwards. Conversely, positive values mean that the mouse has been moved rightwards. Here is an example:

```
; *** Mouse speed
; *** Filename - MouseXSpeed.bb2

BLITZ
BitMap 0,320,256,1
```

```
Slice 0,44,1
Show 0
BitMapOutput 0
Mouse On
MouseArea 0,0,320,256
Repeat
  Locate 0,0:Print "X speed: ",MouseXSpeed,"  "
  Locate 0,1:Print "Y speed: ",MouseYSpeed,"  "
  VWait 5
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

Note that MOUSEXSPEED should only be used after the execution of VWAIT, or during a vertical blank interrupt (#5).

MOUSEYSPEED

```
Mode(s):  Blitz
Function: return the current vertical direction of mouse movement
Syntax:   ydirection=MouseYSpeed
```

If Blitz mode mouse reading has been enabled, MOUSEYSPEED can be used to return the current vertical speed of mouse movement. If a negative value is returned, then the mouse has been moved upwards. If a positive value is returned, the mouse has been moved downwards. See previous example.

MOUSEWAIT

```
Mode(s):   Amiga/Blitz
Statement: wait for click of left mouse button
Syntax:    MouseWait
```

MOUSEWAIT halts program flow until the left mouse button is clicked. This is often useful in Blitz Basic to prevent a program from terminating too quickly and returning you to the editor. Try the following example:

```
; *** Waiting room
; *** Filename - MouseWait.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
BitMapOutput 0
NPrint "Game Over - Press left mouse button"
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

MOUSEWAIT should normally only be used ofor program testing purposes as it severely slows down multi-tasking.

5.4.1 The mouse pointer

If, like me, you hate WIMP (Windows, Icons, Menus and Pointers to the boffins) then you'll be glad to know that you can change the shape of the pointer. The Amiga's mouse pointer is boring. Sorry to offend any hardened pointer-spotters, but a red white and black arrow is hardly indicative of the Amiga's graphical prowess. How about a splash of real colour?

POINTER

```
Mode(s):   Blitz
Statement: attach a sprite to the mouse pointer
Syntax:    Pointer SPRITE#,CHANNEL
```

The POINTER command can be used to dress the mouse pointer in Sunday best. In theory, to change the shape of the pointer arrow, you use the POINTER command followed by the sprite and channel numbers. However, in practise you must execute the following sequence:

1. Load a suitable sprite
2. Create a Slice
3. Execute MOUSE On
4. Execute POINTER

For example:

```
; *** Point me in the right...
; *** Filename - Pointer.bb2

LoadShape 0,"pointer_sprite"
LoadPalette 0,"pointer_sprite",16
GetaSprite 0,0
BLITZ
Bitmap 0,320,DispHeight,4
Slice 0,44,4
Use Palette 0
Show 0
Mouse On
Pointer 0,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

For more information on sprites and their use, please consult Chapter 8.

MOUSEAREA

```
Mode(s):   Blitz
Statement: limit mouse pointer to part of the display
Syntax:    MouseArea X1,Y1,X2,Y2
```

MOUSEAREA is one of those cunning commands whose use is best described by an analogy. Imagine if you would, a little mouse (the fury kind) running freely about the house. MOUSEAREA is rather like a cage, which keeps the mouse from roaming freely. The command creates a rectangular area in which the mouse pointer can move, but cannot move out of. For example:

```
; *** MouseArea example1
; *** Filename - MouseArea1.bb2

MouseArea 10,10,100,100
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

If you need to free the mouse from its cage then simply increase the size of its play area:

```
; *** MouseArea example2
; *** Filename - MouseArea.bb2

MouseArea 0,0,320,200 ; *** This is the default area
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Who needs cats?

5.5 File access

This section will teach you about all aspects of file access. Note that none of these commands are available in Blitz mode.

5.5.1 File requesters

File requesters are used to select files from within simple and complex disk structures (ie. directories and sub-directories).

FILEREQUEST$

```
Mode(s):  Amiga
Function: open a file requester
Syntax:   f$=FileRequest$("TITLE","PATHNAME","FILENAME")
```

The FILEREQUEST$ function opens a standard Amiga-style file requester on the currently used screen. Program flow will halt until the user either selects a file, or hits the requester's "CANCEL" button. If a file was selected, FILEREQUEST$ will return the full name as a string. If "CANCEL" was selected then a null string ("") is returned.

The TITLE$ parameter may be any text string to be used as a title for the file requester. PATHNAME is a string with a maximum length of at least 160. FILENAME is a string with a maximum length of at least 64. The PATHNAME and FILENAME parameters must be set with the MAXLEN statement before a file requester is opened. Try the following example:

```
; *** FileRequest$ example
; *** Filename - FileRequest$.bb2

Screen 0,3+8
ScreensBitMap 0,0
BitMapOutput 0
; *** Maximum length of path and filename
MaxLen PATH$=160
MaxLen FILENAME$=64
; *** Create file requester
A$=FileRequest$("Select a file",PATH$,FILENAME$)
Locate 0,5
; *** Output selected file
NPrint A$
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

5.5.2 Opening a file

OPENFILE

```
Mode(s):  Amiga
Function: open a file
Syntax:   o=OpenFile(FILE#,"FILENAME")
```

OPENFILE is used to open both sequential and random access files. If the file is successfully opened then OPENFILE returns (-1), otherwise (0) is returned. OPENFILE can be used to both read from and write to files. If "FILENAME" does not exist then it will be created by OPENFILE. For example:

```
; *** OpenFile example
; *** Filename - OpenFile.bb2

; *** Save file to RAM disk
If OpenFile(0,"RAM:FILE")
  MaxLen ASTRING$=32
  Fields 0,ANUMBER,ASTRING$
  ANUMBER=Int(Rnd(10)+1)
  ASTRING$="Blitz Basic"
  Put 0,0
  CloseFile 0
  ; *** Read file back into memory
  If OpenFile(0,"RAM:FILE")
    Fields 0,ANUMBER,ASTRING$
    ANUMBER=0
    ASTRING$=""
    Get 0,0
    NPrint ANUMBER
    NPrint ASTRING$
    CloseFile 0
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

5.5.3 Examining files

LOF

```
Mode(s):  Amiga
Function: return the length of a file
Syntax:   l=Lof(FILE#)
```

The LOF function delivers the length of a file in bytes. FILE# is the channel number of the file the function will access. The LOF function can only be used with a file that has previously been opened with OPENFILE. For example:

```
; *** Lof example
; *** Filename - Lof.bb2

If OpenFile(0,"RAM:FILE")
  MaxLen ASTRING$=32
  Fields 0,ASTRING$
  ASTRING$="Douglas"
  Put 0,0
  NPrint Lof(0)," bytes"
  CloseFile 0
  ; *** Wait for a mouse click
  MouseWait
  ; *** Return to Blitz Basic 2 editor
  End
EndIf
```

EXISTS

```
Mode(s):  Amiga
Function: return the length of a file if it exists
Syntax:   e=Exists("FILENAME")
```

This function returns the length of a file. If the file specified in the "FILENAME" parameter does not exist, or a disk is not present in the specified drive, then (0) is returned. For example:

```
; *** Exists example
; *** Filename - Exists.bb2

Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
Locate 0,3
; *** Is Blitz Basic in DF0?
If Exists("DF0:Blitz2")
  NPrint "File exists!"
Else
  NPrint "File does not exist!"
EndIf
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

EOF

```
Mode(s):  Amiga
Function: read the end status of a file
Syntax:   e=Eof(FILE#)
```

The EOF function reads the file data pointer and returns the following values depending on if it has reached the end of the specified file or not. The LOF function can only be used with a file that has previously been opened with OPENFILE:

Table 5.3 : Values returned by EOF

```
End of file?  Return
====================
True           -1
False           0
```

For example:

```
; *** Eof example
; *** Filename - Eof.bb2

If WriteFile (0,"RAM:A FILE")
  ; *** Create file to read
  FileOutput 0
  Print "This is a Blitz Basic file"
  CloseFile 0
  DefaultOutput
  If ReadFile (0,"RAM:A FILE")
    FileInput 0
    ; *** Read file until end is reached
    While Eof(0)=0
      VWait
      Print Inkey$
    Wend
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

LOC

```
Mode(s):  Amiga
Function: return position in a file
Syntax:   l=Loc(FILE#)
```

LOC returns the current position of the data pointer in a file. When a file is first opened, the data pointer is located at position (0). For example:

```
; *** Loc example
; *** Filename - Loc.bb2

If WriteFile (0,"RAM:TESTER")
  ; *** Create file
  FileOutput 0
  Print "Hello from Blitz Basic 2!"
  CloseFile 0
  DefaultOutput
  ; *** Read file
  If ReadFile (0,"RAM:TESTER")
    FileInput 0
    NPrint Edit$(40)
    Print "File length = ",Loc(0)," characters"
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

5.5.4 Deleting files

KILLFILE

```
Mode(s):   Amiga
Statement: delete a file
Syntax:    KillFile "FILENAME"
```

KILLFILE is a rather sinister-sounding command whose purpose is to erase a file from disk. Do be warned that any killed file cannot be replaced, so only kill unimportant data! Here is an example:

```
; *** KillFile example
; *** Filename - KillFile.bb2

If WriteFile (0,"RAM:KILLER")
  ; *** Create file
  FileOutput 0
  Print "I will not exist!"
  CloseFile 0
  DefaultOutput
  ; *** Delete file
  KillFile "RAM:KILLER"
EndIf
```

```
; *** Return to Blitz Basic 2 editor
End
```

5.5.5 Sequential files

Sequential files are those that allow you to read the contents of a file only in the order in which it was originally created. To alter the contents of a sequential file you have to load the entire file into memory, alter the information, and save the whole file back to disk.

READFILE

```
Mode(s):  Amiga
Function: open an existing file for sequential reading
Syntax:   r=ReadFile(FILE#,"FILENAME")
```

The READFILE statement opens an already existing file, specified by "FILENAME", for sequential reading. If the file was successfully opened then (-1) is returned, otherwise (0) is returned. For example:

```
; *** ReadFile example
; *** Filename - ReadFile.bb2

If WriteFile (0,"RAM:A FILE")
  ; *** Create file
  FileOutput 0
  Print "Hello from Blitz Basic!"
  CloseFile 0
  DefaultOutput
  ; *** Read file
  If ReadFile (0,"RAM:A FILE")
    FileInput 0
    NPrint Edit$(40)
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

WRITEFILE

```
Mode(s):  Amiga
Function: create a new file for sequential writing
Syntax:   w=WriteFile(FILE#,"FILENAME")
```

The WRITEFILE statement creates a new file, specified by "FILENAME", for the purpose of sequential file writing. If the file was successfully opened then (-1) is returned, otherwise (0) is returned. For example:

```
; *** WriteFile example
; *** Filename - WriteFile.bb2

If WriteFile (0,"RAM:FILE")
  ; *** Create file
  FileOutput 0
  Print "WriteFile example"
  CloseFile 0
  DefaultOutput
  ; *** Read file
  If ReadFile (0,"RAM:FILE")
    FileInput 0
    NPrint Edit$(40)
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

FILEOUTPUT

```
Mode(s):   Amiga/Blitz
Statement: cause print commands to output to sequential file
Syntax:    FileOutput FILE#
```

FILEOUTPUT is used to send all future print commands to the specified sequential file (FILE#). Upon file closure, printing should be directed to another output channel.

FILEINPUT

```
Mode(s):   Amiga/Blitz
Statement: cause input commands to receive from sequential file
Syntax:    FileInput FILE#
```

FILEINPUT is used to cause all future input commands to receive from the specified sequential file (FILE#). Upon file closure, input should be directed to another input channel.

Here is an example which demonstrates the use of FILEINPUT and FILEOUTPUT:

```
; *** FileOutput/Input example
; *** Filename - FileOutput.bb2

If WriteFile (0,"RAM:INOUT")
  FileOutput 0
  Print "A load of rubbish!"
  CloseFile 0
  DefaultOutput
  If ReadFile (0,"RAM:INOUT")
    FileInput 0
    NPrint Edit$(40)
    CloseFile 0
    DefaultInput
    ; *** Wait for a mouse click
    MouseWait
    ; *** Return to Blitz Basic 2 editor
    End
  EndIf
EndIf
```

FILESEEK

```
Mode(s):   Amiga
Statement: move to a point in the specified file
Syntax:    FileSeek FILE#,POSITION
```

The FILESEEK statement can be used to moves to a particular point in the specified file (FILE#). The POSITION parameter must be less than the length of the file. For example:

```
; *** FileSeek example
; *** Filename - FileSeek.bb2

If WriteFile (0,"RAM:FILE")
  ; *** Create file
  FileOutput 0
  Print "The best BASIC is Blitz "
  CloseFile 0
  If OpenFile (0,"RAM:FILE")
    ; *** Search for end of file
    FileSeek 0,Lof(0)
    ; *** Add word to file
    NPrint "Basic!"
    CloseFile 0
    DefaultOutput
    ; *** Read new file
    If ReadFile (0,"RAM:FILE")
      FileInput 0
      NPrint Edit$(80)
      ; *** Wait for a mouse click
      MouseWait
    EndIf
  EndIf
EndIf
; *** Return to Blitz Basic 2 editor
End
```

CLOSEFILE

```
Mode(s):   Amiga
Statement: close a file
Syntax:    CloseFile FILE#
```

CLOSEFILE closes the file specified by FILE#. try the following example:

```
; *** CloseFile example
; *** Filename - CloseFile.bb2

If WriteFile (0,"RAM:FILEOFAX")
  FileOutput 0
  Print "A closed case"
  CloseFile 0
EndIf
; *** Return to Blitz Basic 2 editor
End
```

5.5.6 Random access files

The most obvious difference between a random access file and a sequential file is in the access method. With a sequential file the entire file must be loaded into memory in order to access one field. In a random access file, however, one record can be read into memory without having to read in the entire file. The disadvantage of random access files is that a larger file area is required on disk.

FIELDS

```
Mode(s):   Amiga/Blitz
Statement: set-up fields of a random access file record
Syntax:    Fields FILE#,VAR1[,VAR2...]
```

The FIELDS statement is used to set-up the fields of a random access file record. The numeric expression FILE# is the number of the data channel of a data file previously opened with OPENFILE. The VAR parameters specify a list of variables that can be read from or written to the file.

Any string variables in this list must have been initialised to contain a maximum number of characters using the MAXLEN statement.

PUT

```
Mode(s):   Amiga
Statement: Write a specific record to a random access file.
Syntax:    Put FILE#,RECORD
```

PUT writes a specific record to a random access file.

GET

```
Mode(s):   Amiga
Statement: Read a specific record from a random access file
Syntax:    Get FILE#,RECORD
```

GET reads a specific record from a random access file.

The following example demonstrates the use of the FIELDS, GET, and PUT statements in the creation of random access files:

```
; *** Random access file example ; *** Filename - Random_Access.bb2


If OpenFile (0,"RAM:TEST")
  ; *** Maximum length of string field
  MaxLen B$=32
  ; *** Define fields
  Fields 0,A,B$
  ; *** Field contents
  A=17
  B$="Blitz Basic"
  Put 0,0
  CloseFile 0
  ; *** Read file
  If OpenFile (0,"RAM:TEST")
    ; *** Define fields
    Fields 0,A,B$
    ; *** Initialise variables (not necessary)
    A=0
    B$=""
    ; *** Grab variables from file
    Get 0,0
    NPrint "A = ",A
    NPrint "B$ = ",B$
    CloseFile 0
    ; *** Wait for a mouse click
    MouseWait
  EndIf
EndIf
; *** Return to Blitz Basic 2 editor
End
```

5.5.7 Advanced file access

The following commands are primarily of use to the advanced Blitz Basic programmer. If you don't know what you're doing, then hands off!

DOSBUFFLEN

```
Mode(s):   Amiga/Blitz
Statement: set file buffer
Syntax:    DosBuffLen BYTES
```

The DOSBUFFLEN statement controls the Blitz Basic file handling buffer. Initially, each file is allocated a 2048 byte buffer, however this may be decreased if memory is tight. For example:

```
; *** DosBuffLen example
; *** Filename - DosBuffLen.bb2

DosBuffLen 2000
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CATCHDOSERRORS

```
Mode(s):   Amiga/Blitz
Statement: force DOS errors to report on a Blitz window
Syntax:    CatchDosErrors
```

CATCHDOSERRORS is used to force AmigaDOS I/O errors into opening on a Blitz Basic window, as opposed to the Workbench screen. Try the following example:

```
; *** CatchDosErrs example
; *** Filename - CatchDosErrs.bb2

Screen 0,3
Window 0,0,12,320,DispHeight-12,$1008,"Window",1,2
; *** Send errors to window
```

```
CatchDosErrs
; *** Try reading file
If ReadFile (0,"DF0:GARBAGE")
Else
  Print "Can't open file"
EndIf
Repeat
Until WaitEvent=$200
; *** Return to Blitz Basic 2 editor
End
```

READMEM

```
Mode(s):   Amiga
Statement: read a number of bytes into an absolute memory location
Syntax:    ReadMem FILE#,ADDRESS,LENGTH
```

The READMEM statement reads a number of bytes, determined by the LENGTH parameter, into an absolute memory location, specified by the ADDRESS parameter, from a file. FILE# is the number of a file already opened with OPENFILE.

WRITEMEM

```
Mode(s):   Amiga
Statement: write a number of bytes from an absolute memory location
Syntax:    WriteMem FILE#,ADDRESS,LENGTH
```

The WRITEMEM statement writes a number of bytes, determined by the LENGTH parameter, from an absolute memory location, specified by the ADDRESS parameter, to a file. FILE# is the number of a file already opened with OPENFILE.

5.6 End-of-Chapter summary

Text can be printed onto the screen using PRINT and NPRINT. NPRINT automatically outputs a newline character.

The text style can be altered using LOADBLITZFONT. LOADBLITZFONT can only be used with eight-by-eight non-proportional fonts.

The COLOUR statement is used to alter the colour used to render text to BitMaps.

LOCATE can be used to position the text cursor.

The CURSOR statement is used to alter the appearance of the text cursor.

Blitz Basic provides full control over the Amiga keyboard. The keyboard must be correctly enabled to be read in Blitz mode.

The appearance of the mouse pointer can be changed with POINTER. Blitz Basic also provides full control over standard nine pin joysticks.

There are two types of file access: sequential and random access. With sequential files the entire file must be loaded into memory in order to access one field. In random access files, however, one record can be read into memory without having to read in the entire file.

Chapter 6 : BitMaps and Slices

This chapter explains how BitMaps and Slices are created and manipulated. It will also show you how to create smooth-scrolling and dual-playfield displays.

6.1 Creating a BitMap

BitMap objects, or BitMaps, are used for the rendering of graphics. Nearly all of the Blitz Basic 2 graphic commands require a BitMap to output onto, with the notable exceptions being the window and sprite commands (more on those later).

BitMaps can either be created from scratch by the BITMAP statement, or borrowed from a convenient screen using SCREENSBITMAP.

BITMAP

```
Mode(s):   Amiga/Blitz
Statement: open a new BitMap
Syntax:    BitMap BITMAP#,WIDTH,HEIGHT,BITPLANES
```

This statement creates and initializes a BitMap (BITMAP#). The WIDTH and HEIGHT parameters specify the dimensions of the BitMap in pixels. The BITPLANES parameter is the number of bitplanes associated with the BitMap. The value you specify (ranging from one to six) determines the number of colours that can be displayed on the BitMap, as shown in the following table:

Table 6.1 : Number of colours per bitplane

```
Bitplanes  Colours
==================
1          2
2          4
3          8
4          16
5          32
6          64
```

Here are some examples:

```
; *** BitMap example
; *** Filename - BitMap.bb2

BLITZ
BitMap 0,320,256,1 ; *** 2 colour BitMap
Slice 0,44,1
Show 0
```

```
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** BitMap example 2
; *** Filename - BitMap2.bb2

BLITZ
BitMap 0,640,256,5 ; *** Double-width 32 colour BitMap
Slice 0,44,5
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** BitMap example 3
; *** Filename - BitMap3.bb2

BLITZ
BitMap 0,320,256,3 ; *** 8 colour BitMap
BitMapOutput 0
Slice 0,44,3
Show 0
Locate 15,10
NPrint "A BitMap"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.1.2 Manipulating BitMaps

USE BITMAP

```
Mode(s):   Amiga/Blitz
Statement: set current BitMap
Syntax:    Use BitMap BITMAP#
```

USE BITMAP is used to set a specified BitMap as the current BitMap. For example:

```
; *** Use BitMap example
; *** Filename - Use BitMap.bb2

BitMap 0,320,256,3
For A=1 To 100
  Plot Rnd(320),Rnd(256),Rnd(6)+1
Next A
BitMap 1,320,256,3
BLITZ
Slice 0,44,3
For A=1 To 10
  Show MAP : MAP=1-MAP: Use BitMap MAP
VWait 30
Next A
; *** Return to Blitz Basic 2 editor
End
```

FREE BITMAP

```
Mode(s):   Amiga/Blitz
Statement: erase a BitMap
Syntax:    Free BitMap BITMAP#
```

The FREE BITMAP statement closes a BitMap and frees any memory occupied it. For example:

```
; *** Free BitMap example
; *** Filename - Free BitMap.bb2

BitMap 0,320,256,3
For A=1 To 100
  Circle Rnd(320),Rnd(256),Rnd(10)+2,Rnd(6)+1
Next A
BLITZ
Slice 0,44,3
Show 0
VWait 100
Free BitMap 0
; *** Return to Blitz Basic 2 editor
End
```

COPYBITMAP

```
Mode(s):   Amiga/Blitz
Statement: clone a BitMap
Syntax:    CopyBitMap SOURCE,DESTINATION
```

This statement makes a carbon copy of a BitMap. SOURCE is the number of the BitMap to clone and DESTINATION is the number of the destination BitMap. Try the following example:

```
; *** CopyBitMap example
; *** Filename - CopyBitMap.bb2

BitMap 1,320,256,3
BitMap 0,320,256,3
BitMapOutput 0
BLITZ
Slice 0,44,3
Show 0
For A=1 To 100
  Locate Rnd(25)+3,Rnd(25)
  Colour Rnd(6)+1
  Print "Game Over"
VWait
Next A
CopyBitMap 0,1
VWait 20
Cls 0
VWait 50
Use BitMap 1
Show 1
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SCREENSBITMAP

```
Mode(s):   Amiga/Blitz
Statement: attach a BitMap to an intuition screen
Syntax:    ScreensBitMap SCREEN#,BITMAP#
```

Blitz Basic also allows the user to "attach" a BitMap to an Intuition Screen. BitMaps are automatically created when these Screens are opened. For example:

```
; *** ScreensBitMap example
; *** Filename - ScreensBitMap.bb2

BitMap 0,320,256,3
PalRGB 0,0,0,0,0
Screen 0,3,"Stardom"
ScreensBitMap 0,0
Use Palette 0
For A=1 To 100
  Plot Rnd(320),Rnd(200)+30,Rnd(6)+1
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SCROLL

```
Mode(s):   Amiga/Blitz
Statement: move a portion of a BitMap
Syntax:    Scroll X1,Y1,WIDTH,HEIGHT,X2,Y2[,BITMAP#]
```

This statement allows you to move, or scroll, a rectangular portion of a BitMap. X1 and Y1 are the co-ordinates of the upper left-hand corner of the rectangle and WIDTH and HEIGHT specify its size. The X2 and Y2 parameters are the destination co-ordinates. If the optional BITMAP# parameter is included then the rectangle is taken from this BitMap instead, and copied to the current BitMap. Here's an example:

```
; *** Scroll example
; *** Filename - Scroll.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
For A=1 To 50
  Circlef Rnd(320),Rnd(100),Rnd(20)+10,Rnd(5)+1
Next A
Scroll 0,0,320,100,0,140
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

REMAP

```
Mode(s):   Amiga/Blitz
Statement: change pixels of one colour to another colour
Syntax:    ReMap COLOUR1#,COLOUR2#[,BITMAP#]
```

The REMAP statement can change pixels of one colour on a BitMap to another colour. COLOUR1# specifies the colour to change and COLOUR2# is the number of the new colour. If the optional BITMAP# parameter is included then the a BitMap other than the current BitMap may be used. Try the following example:

```
; *** ReMap example
; *** Filename - ReMap.bb2

BLITZ
BitMap 0,320,256,5
Slice 0,44,5
Show 0
; *** Plot a boring white starfield
For COLS=1 To 14
  RGB COLS,15,15,15
Next COLS
For A=0 To 300
  Plot Rnd(320),Rnd(256),Rnd(14)+1
Next A
MouseWait
; *** Add a splash of colour
For B=1 To 14
  ReMap B,B+8
Next B
Repeat : Until Joyb(0)=0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

BITPLANESBITMAP

```
Mode(s):   Amiga/Blitz
Statement: create new BitMap with bitplanes from old BitMap
Syntax:    BitPlanesBitMap SOURCE,DESTINATION,BITPLANES
```

This statement is used to create a "dummy" BitMap (DESTINATION) from the source BitMap (SOURCE), with only the bitplanes specified by the BITPLANES parameter. This is useful for increasing blitting speed because of the fewer bitplanes involved.

Table 6.2 : The BITPLANES parameter

```
Bitplane  Flag
==============
1         $01
2         $02
3         $04
4         $08
5         $10
6         $20
7         $40
8         $80
```

Flags can be combined with the logical (|) operator.

The BITPLANESBITMAP statement can also be used to create special effects, such as shadows. This example was created by Tim Caldwell:

```
; *** BitPlanesBitMap example
; *** Filename - BitPlanesBitMap.bb2

BLITZ
BitMap 0,320,256,5
BitMapOutput 0
; *** Create dummy BitMap (bitplane 5)
BitPlanesBitMap 0,1,$10
Slice 0,44,5
Show 0
X=80 : Y=48 : W=160 : H=160
Use BitMap 0
; *** Draw BitMap graphics
For COL=0 To 15
  R=QLimit(Red(COL)-5,0,15)
  G=QLimit(Green(COL)-5,0,15)
  B=QLimit(Blue(COL)-5,0,15)
  RGB COL+16,R,G,B
  Boxf X,Y,X+W,Y+H,COL
  X+4 : Y+4 : W-8 : H-8
Next COL
X=120 : Y=88 : W=80 : H=80
; *** Use dummy BitMap
Use BitMap 1
While Joyb(1)=0
  ; *** Use joystick to move shadow
  JX=Joyx(1) : JY=Joyy(1)
  If JX OR JY=True
    Cls
    X=QLimit(X+JX,0,320-W)
    Y=QLimit(Y+JY,0,256-H)
  EndIf
  ; *** Draw shadow
  Boxf X,Y,X+W,Y+H,1
  VWait
Wend
; *** Return to Blitz Basic 2 editor
End
```

6.1.3 Loading and saving BitMaps

LOADBITMAP

```
Mode(s):   Amiga
Statement: load an IFF screen from disk
Syntax:    LoadBitMap BITMAP#,"FILENAME"[,PALETTE#]
```

The lOADBITMAP statement loads an IFF picture (such as a DPaint file) into a previously opened BitMap. If the optional PALETTE parameter is included then the picture's palette may be loaded into a palette object. Here is an example:

```
; *** LoadBitMap example
; *** Filename - LoadBitMap.bb2

BitMap 0,320,256,5
LoadBitMap 0,"FILENAME.IFF",0
BLITZ
Slice 0,44,5
Show 0
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SAVEBITMAP

```
Mode(s):   Amiga
Statement: save an IFF screen to disk
Syntax:    SaveBitMap BITMAP#,"FILENAME"[,PALETTE#]
```

SAVEBITMAP saves a BitMap to disk as an IFF file. If the optional PALETTE# parameter is included then the picture's palette may be saved to disk as well:

```
; *** SaveBitMap example
; *** Filename - SaveBitMap.bb2

BitMap 0,320,256,5
; *** Draw a nice random picture
For A=1 To 100
  Circlef Rnd(320),Rnd(256),Rnd(10)+5,Rnd(10)+5,Rnd(6)+1
Next A
BLITZ
Slice 0,44,5
Show 0
; *** Pop into Amiga mode and save BitMap
QAMIGA
SaveBitMap 0,"df0:Elipse.IFF"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.1.4 Display synchronisation

The computer display is updated fifty times every second on PAL systems, and sixty times a second on NTSC systems. Here, the display is created by an electron beam which scans across every line of the screen until it reaches the bottom right-hand corner, where it jumps back to the top of the screen again. The period between the completion of one display cycle and the next is known as the "vertical blank period".

Because some Blitz commands work faster than others, it is often useful to wait for the next vertical blank period before executing them, so as to achieve perfect display synchronisation. This is where the VWAIT statement comes in.

VWAIT

```
Mode(s):   Amiga/Blitz
Statement: wait for next vertical blank period
Syntax:    VWait [FRAMES]
```

This statement waits for the next vertical blank period and is used to achieve perfect display synchronisation. The optional FRAMES parameter may be used to specify a particular number of vertical blanks (the default is one). Try the following example which illustrates the use of VWAIT:

```
; *** VWait example
; *** Filename - VWait.bb2

; *** Pop onto Blitz mode
BLITZ
; *** Create a Blitz mode display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Create a shape
Boxf 10,10,50,50,5
GetaShape 0,10,10,50,50
Cls
; *** Initialize BBLIT buffer
Buffer 0,16384
; *** Flickery animation
For X=1 To 250
  UnBuffer 0
  BBlit 0,0,X,50
Next X
VWait 50
; *** No flicker!
For X2=1 To 250
  VWait
  UnBuffer 0
  BBlit 0,0,X2,50
Next X2
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

VPOS

```
Mode(s):  Amiga/Blitz
Function: return the video beam's vertical position
Syntax:   v=VPos
```

VPOS returns the video beam's vertical position. This is primarily of use in high-speed animations where screen update needs to by syncronised to a certain video beam position (not the top of the frame as with VWAIT). However, it can also be used as a high-speed random number generator, as in the following example:

```
; *** VPos example ** Filename - VPos.bb2
; *** Loop 20 times
For A=1 To 20
  ; *** Return video beam position
  RANDOM=VPos
  ; *** Output returned value
  NPrint RANDOM
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.2 Defining a Slice

Slices are Blitz mode screens. However, unlike screens, Slices can be used to create dual-playfield displays (more on these later), smooth scrolling, double buffering and more!

A Slice description includes information on display mode, palette and sprite and bitplane details.

A Slice's x co-ordinate is calculated in a way which causes the Slice to be horizontally centred based on its width.

More than one Slice may be set up at a time, allowing different areas of the display to take on different properties:

```
The SHOW statement is used to display a BitMap in a Slice.
```

There are limits placed upon how multiple Slices may be arranged. Multiple Slices must be positioned vertically on top of each other, with a gap of two horizontal lines between each Slice. Slices must not overlap or be positioned together on the x-axis.

SLICE

```
Mode(s):   Amiga/Blitz
Statement: create a Slice object
Syntax:    Slice SLICE#,Y,FLAGS1
Syntax 2:  Slice SLICE#,Y,W,H,FLAGS2,D,S,COLS,WIDTH1,WIDTH2
```

6.2.1 Syntax 1

The Slice statement is used to define a Slice object. SLICE# is the number of the Slice to be defined. The Y parameter specifies the vertical location of the top of the Slice, ranging from 44 to the bottom of the current display. In other words, a value of 44 displays the Slice at the very top of a display.

In the first syntax, FLAGS1 refers to the number of bitplanes to be shown in the Slice, from one (a maximum of two colours) to six (a maximum of 64 colours). This syntax automatically creates a low-resolution Slice, however by adding eight to the FLAGS1 parameter this may be changed to a high-resolution Slice.

Table 6.3 : The FLAGS1 parameter

FLAGS1	Resolution	Width	Bitplanes	Colours
1	Low	320	1	2
2	Low	320	2	4
3	Low	320	3	8
4	Low	320	4	16
5	Low	320	5	32
6	Low	320	6	64 (Half-Brite)
9	High	640	1	2
10	High	640	2	4
11	High	640	3	8
12	High	640	4	16

Note that the height of a Slice set up with the first syntax will be 256 pixels on a PAL Amiga, or 200 pixels on an NTSC Amiga.

Here are some examples:

```
; *** Slice example ** Filename - Slice1.bb2

BitMap 0,320,256,9
BLITZ
Slice 0,44,1 ; *** 2 colour hi-res Slice
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Slice example 2 ** Filename - Slice2.bb2

BitMap 0,320,256,5
BLITZ
Slice 0,44,5 ; *** 32 colour low-res Slice
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.2.2 Syntax 2

W and H specify the width and height (in pixels) of the Slice. D, or DEPTH, is the number of bitplanes to be shown in the Slice. The S parameter specifies the number of available sprite channels. Each Slice can have up to eight sprite channels.

The WIDTH1 and WIDTH2 parameters specify the width, in pixels, of any BitMaps to be shown in the Slice. If a dual-playfield Slice is created then WIDTH1 refers to the width of the foreground BitMap and WIDTH2 refers to the background BitMap. Otherwise, both WIDTH1 and WIDTH2 should be set the same. These parameters allow you to display super-BitMaps (those larger than the physical display).

The FLAGS2 parameter is used to customise the Slice to your every requirements.

Table 6.4 : The FLAGS2 parameter

```
FLAGS2  Slice                           Maximum bitplanes
=========================================================
$fff8   Low-resolution                  6
$fff9   High-resolution                 4
$fffa   Low-resolution, dual-playfield  6
$fffb   High-resolution, dual-playfield 4
$fffc   HAM-mode                        6
```

Here are some examples:

```
; *** Slice example 3
; *** Filename - Slice3.bb2

BitMap 0,320,256,3
BLITZ
; *** 8 colour low-res Slice
Slice 0,44,320,256,$fff8,3,8,8,320,320
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Slice example 4
; *** Filename - Slice4.bb2

BitMap 0,320,256,1
BLITZ
; *** 2 colour hi-res Slice
Slice 0,44,320,256,$fff9,1,8,2,320,320
Show 0
BitMapOutput 0
Print "Hello"
```

```
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.2.3 Manipulating Slices

USE SLICE

```
Mode(s):   Amiga/Blitz
Statement: set current Slice
Syntax:    Use Slice SLICE#
```

USE SLICE is used to set the currently used Slice. This allows you to direct all Slice manipulating commands to the specified Slice number:

```
; *** Use Slice example
; *** Filename - Use Slice.bb2

Use Slice 1
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FREESLICES

```
Mode(s):   Amiga/Blitz
Statement: erase all Slices in use
Syntax:    FreeSlices
```

Use the FREESLICES command to free all Slices currently in use. For example:

```
; *** FreeSlices example
; *** Filename - FreeSlices.bb2

; *** Open a BitMap
BitMap 0,320,256,3
; *** Create some BitMap graphics
For A=1 To 100
  Circlef Rnd(320),Rnd(256),Rnd(10)+2,Rnd(6)+1
Next A
; *** Pop into Blitz mode
BLITZ
```

```
; *** Create a slice
Slice 0,44,320,256,$fff8,3,8,8,320,320
; *** Display BitMap graphics in slice
Show 0
; *** Pause briefly
VWait 100
; *** Remove old slice
FreeSlices
; *** Create another slice
Slice 0,44,320,256,$fff9,3,8,8,320,320
; *** Display BitMap graphics in slice
Show 0
VWait 100
; *** Remove old slice (again!)
FreeSlices
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SETBPLCON0

```
Mode(s):   Amiga/Blitz
Statement: set Slice display mode
Syntax:    SetBPLCON0 DEFAULT
```

This statement allows advanced control of Slice display modes. The DEFAULT parameter should be set as follows:

Table 6.5 : Display modes

```
BIT  Mode
=========================================
1    External sync (for genlock enabling)
2    Interlace mode
3    Enable light pen
```

Here is an example:

```
; *** SetBPLCON0 example
; *** Filename - SetBPLCON0.bb2

; *** Create a BitMap (4 bitplanes)
BitMap 0,640,512,4
; *** Set Interlace mode
SetBPLCON0 4
; *** Pop into Blitz mode
```

```
BLITZ
; *** Open large slice
Slice 0,44,640,256,$fffb,4,8,8,1280,1280
; *** Declare interrupt
SetInt 5
  If Peek($dff004)<0 Show 0,0,0 Else Show 0,0,1
End SetInt
; *** Output BitMap graphics
For A=1 To 400
  Circle Rnd(640),Rnd(512),Rnd(30)+10,Rnd(16)
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.2.4 Displaying a BitMap in a Slice

SHOW

```
Mode(s):   Amiga/Blitz
Statement: display a BitMap in the current Slice
Syntax:    Show BITMAP#[,X,Y]
```

The SHOW statement is used to display a BitMap in the currently used Slice. If the optional X and Y parameters are included then the BitMap is positioned at these co-ordinates. For example:

```
; *** Show example
; *** Filename - Show.bb2

; *** Number of stars to plot
STARS=100
; *** Pop into Blitz mode
BLITZ
; *** Open a BitMap (2 bitplanes)
BitMap 0,320,DispHeight,2
; *** Plot a random starfield
For A=0 To STARS
  Plot Rnd(320),Rnd(DispHeight),Rnd(3)+1
Next A
; *** Create a slice
Slice 0,44,2
; *** Grab BitMap's palette
Use Palette 0
; *** Display BitMap
Show 0
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

If the BitMap is physically larger than the Slice then the SHOW statement may be used to scroll the BitMap about the display.

Here is an example:

```
; *** Land generator
; *** Filename - Show2.bb2

; *** Nip into Blitz mode
BLITZ
; *** Open 2-screen wide display
BitMap 0,640,256,2
Slice 0,44,320,256,$fff8,2,8,4,640,640
Show 0
; *** Simple colour graduation
For A=0 To 15
  ColSplit 1,0,A,A,A*17
  ColSplit 3,A,A,A,100+A*17
Next
Cls 1
; *** Draw mountain landscape
Y=200 : LAND=3 : DI=-1
For X=0 To 640
  D=Int(Rnd(LAND))
  If D=1 Then DI=-1
  If D=2 Then DI=1
  Let Y+DI
  If Y<0 Then Y=0
  If Y>256-1 Then Y=255
  Line X,256,X,Y,3
Next X
; *** Scroll landscape
For A=1 To 320
  Show 0,A,0
VWait
Next A
; *** Return to Blitz Basic 2 editor
End
```

Do not use SHOW for dual-playfield Slices. Use the following commands instead.

SHOWF

```
Mode(s):   Amiga/Blitz
Statement: display a BitMap in the foreground of the current Slice
Syntax:    ShowF BITMAP#[,X,Y]
Syntax 2:  ShowF BITMAP#,X,Y,ShowB X2
```

The SHOWF statement is used to display a BitMap in the foreground of the currently used dual-playfield Slice. If the optional X and Y parameters are included then the BitMap is positioned at these coordinates. The optional SHOWB X2 parameter (syntax 2) is of use when a Slice has been set up to display a foreground BitMap only. In this case, the x offset of the background BitMap should be specified by the SHOWB parameter.

SHOWB

```
Mode(s):   Amiga/Blitz
Statement: display a BitMap in the background of the current Slice
Syntax:    ShowB BITMAP#[,X,Y]
Syntax 2:  ShowB BITMAP#,X,Y,ShowF X2
```

The SHOWB statement is used to display a BitMap in the background of the currently used dual-playfield Slice. If the optional X and Y parameters are included then the BitMap is positioned at these coordinates. The optional SHOWF X2 parameter (syntax 2) is of use when a Slice has been set up to display a background BitMap only. In this case, the x offset of the foreground BitMap should be specified by the SHOWF parameter:

```
; *** Dual Playfield example
; *** Filename - ShowF.bb2

BLITZ
; *** Open 2 BitMaps
BitMap 0,352,256+32,2
BitMap 1,352,256+32,2
; *** Create single Slice to house BitMaps
Slice 0,44,320,256,$fffa,4,8,32,352,352
; *** Display BitMap 0 in background
ShowB 0
; *** Display BitMap 1 in foreground
ShowF 1
RGB 1,0,0,15
RGB 9,15,0,0
Use BitMap 0
; *** Draw foreground graphics
For Y=0 To 256 Step 16
  For X=0 To 352 Step 16
    COL=1-COL
    Boxf X,Y,X+16,Y+16,COL
  Next
Next
Use BitMap 1
; *** Draw background graphics
For Y=0 To 288 Step 16
  For X=0 To 352 Step 16
    COL=1-COL
    Boxf X,Y,X+16,Y+16,COL
  Next
Next
; *** Scroll playfields
While Joyb(0)=0
  VWait
  X=QWrap(X+1,0,32)
  Y=QWrap(Y+1,0,32)
  ShowB 0,X,0,Y
  ShowF 1,0,Y,X
Wend
; *** Return to Blitz Basic 2 editor
End
```

SHOWBLITZ

```
Mode(s):   Blitz
Statement: redisplay all Slices
Syntax:    ShowBlitz
```

SHOWBLITZ redisplays all of the Slices curently opened. This is primarily of use when you have made a trip into Amiga mode and wish to return to Blitz mode without corrupting any Slices.

DISPLAY

```
Mode(s):   Blitz
Statement: allows you to turn the display on or off
Syntax:    DisplayOn/Off
```

The DISPLAY statement is used to turn the whole display on or off. If DISPLAY is set to OFF then the display will become a solid block of colour 0. Here is an example:

```
; *** Display example
; *** Filename - Display.bb2

BitMap 0,320,256,3
; *** Direct PRINT statements to BitMap
BitMapOutput 0
For A=1 To 100
  ; *** Select a random cursor location...
  Locate Rnd(30),Rnd(25)
  ; *** ...And a random colour
  Colour Rnd(6)+1
  Print "Blitz Basic"
Next A
; *** Enter Blitz mode
BLITZ
; *** Turn off display
DisplayOff
Slice 0,44,3
Show 0
VWait 100
; *** Turn on display
DisplayOn
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

6.3 End-of-Chapter summary

BitMaps are used for rendering graphics and may be created using the BITMAP statement, or borrowed from a screen using the SCREENSBITMAP statement.

Blitz Basic also allows you to load and manipulate BitMaps in the form of IFF graphics.

Slices are Blitz mode screens. However, unlike screens, Slices can be used to create dual-playfield and double-buffered displays.

BitMaps are displayed in Slices using the SHOW statement. The SHOW statement may also be used to create gigantic scrolling displays.

Table 6.6 : BitMap and Slice commands

Command	Function
BITMAP	Create a BitMap
BITPLANESBITMAP	Create a "Dummy" BitMap
COPYBITMAP	Clone a BitMap
DISPLAY	Turn display on or off
FREE BITMAP	Close a BitMap
FREESLICES	Close all Slices
LOADBITMAP	Load an IFF screen
REMAP	Change BitMap colours
SAVEBITMAP	Save an IFF screen
SCREENSBITMAP	Attach BitMap to Intuition screen
SHOW	Display BitMap in a Slice
SHOWB	Display BitMap in background
SHOWBLITZ	Redisplay all Slices
SHOWF	Display BitMap in foreground
SLICE	Create a Slice

```
  USE BITMAP       Set current BitMap
  USE SLICE        Set current Slice
```

Chapter 7 : Graphics

Blitz Basic 2 is a powerful extended BASIC language. This means that it supports commands not present in languages such as AmigaBasic or HiSoft Basic. As well as a comprehensive array of drawing commands, the Blitz programmer also has Colour Palettes, IFF Animation and Copper Lists at their disposal. Read on...

7.1 2D Drawing

Blitz Basic can generate fabulous low-resolution and high-resolution graphic displays using its powerful drawing commands. These graphic displays are made up of small blocks of colour called pixels and all screens are composed of thousands of pixels in varying arrangements. Here's how we manipulate these pixels to produce anything from lines and circles to starfields and megademos.

7.1.1 Clearing with colour

CLS

```
Mode(s):   Amiga/Blitz
Statement: clear a BitMap
Syntax:    Cls [COLOUR]
```

This statement is used to fill the currently used BitMap with the colour specified by the COLOUR parameter. If the optional COLOUR parameter is omitted then the BitMap will be cleared with colour (0). A COLOUR parameter of (-1) will cause the entire BitMap to be inverted. For example:

```
; *** Cls example
; *** Filename - Cls.bb2

; *** Open a screen...
Screen 0,3
; *** ...And grab its BitMap
ScreensBitMap 0,0
; *** Loop until mouse button clicked
While Joyb(0)=0
  ; *** Clear screen a variety of different colours
  Cls Rnd(5)+1
  ; *** Pause briefly
  VWait 10
Wend
; *** Return to Blitz Basic 2 editor
End
```

7.1.2 Gunpowder plot

PLOT

```
Mode(s):   Amiga/Blitz
Statement: plot an individual colour pixel
Syntax:    Plot X,Y,COLOUR
```

The PLOT statement plots a single pixel at coordinates X,Y in colour COLOUR on the currently used BitMap. A COLOUR parameter of (-1) will cause the pixel to be inverted. For example:

```
; *** Plot Starfield
; *** Filename - Plot_Example.bb2

; *** Number of stars in starfield
STARS=100
; *** Nice space-type palette (i.e. grey!)
PalRGB 0,0,0,0,0
PalRGB 0,1,10,10,10
PalRGB 0,2,7,7,7
PalRGB 0,3,3,3,3
; *** Create Blitz mode display
BLITZ
BitMap 0,320,DispHeight,2
; *** Plot a random starfield
For A=0 To STARS
  Plot Rnd(320),Rnd(DispHeight),Rnd(3)+1
Next A
Slice 0,44,320,DispHeight,$fff8,2,8,8,320,320
Use Palette 0
Show 0
; *** Wait for left mouse button
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

You can really only make very simple pictures with PLOT. To make more comlicated ones you need special equipment such as a graphics tablet. Blitz Basic does not support these devices directly, so any graphics should be created using a paint package, such as Deluxe Paint, saved in IFF format and loaded into Blitz using the LOADBITMAP statement.

7.1.3 A few pointers

POINT

```
Mode(s):  Amiga/Blitz
Function: return the colour of an individual pixel
Syntax:   a=Point(X,Y)
```

Use the POINT function to return the colour of a particular pixel on the currently used BitMap. If the chosen coordinates specify a pixel outside the currently defined BitMap then a value of (-1) will be returned. Try the following example:

```
; *** Point me in the right...
; *** Filename - Point_Example.bb2

; *** Define palette
PalRGB 0,0,0,0,0
PalRGB 0,1,10,0,7
; *** Open a screen and grab its BitMap
Screen 0,3
ScreensBitMap 0,0
; *** Direct PRINT statement to BitMap
BitMapOutput 0
Use Palette 0
; *** Draw 1000 coloured boxes
For A=1 To 1000
  X1=Rnd(310)
  X2=X1+10
  Y1=Rnd(DispHeight-20)+15
  Y2=Y1+10
  Boxf X1,Y1,X2,Y2,Rnd(5)
Next A
Locate 0,2
; *** Select random pixel to test
B=Point(Rnd(320),Rnd(DispHeight))
Print B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.1.4 It's a fine line

LINE

```
Mode(s):   Amiga/Blitz
Statement: draw a line
Syntax:    Line X1,Y1,X2,Y2,COLOUR
Syntax 2:  Line X2,Y2,COLOUR
```

The LINE statement draws a line connecting two pixels on the currently used BitMap. The first syntax uses two sets of graphic coordinates to join, followed by the colour of the line. A COLOUR parameter of (-1) will cause the line to be inverted. For example:

```
; *** Line Example
; *** Filename - Line.bb2

; *** Open a screen and grab its BitMap
Screen 0,3
ScreensBitmap 0,0
; *** Draw a simple straight line
Line 10,10,50,10,1
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

If the optional X1 and Y1 parameters are omitted, as in the second syntax, then the current position of the graphics cursor will be used as the starting co-ordinates:

```
; *** More Lines
; *** Filename - Line2.bb2

; *** Open a screen and grab its BitMap
Screen 0,3
ScreensBitmap 0,0
; *** Draw 50 lines at random co-ordinates
For A=1 To 50
  Line Rnd(320),Rnd(DispHeight),Rnd(7)+1
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.1.5 Boxing clever

BOX

```
Mode(s):   Amiga/Blitz
Statement: draw a rectangular outline
Syntax:    Box X1,Y1,X2,Y2,COLOUR
```

Rectangular outlines can be drawn on the currently used BitMap with the BOX statement. X1 and Y1 are the coordinates of the top left-hand corner of the rectangle and X2 and Y2 are the coordinates of the bottom right-hand corner. COLOUR is the colour of the outline; a COLOUR parameter of (-1) will cause the rectangle to be inverted. For example:

```
; *** Boxing ring
; *** Filename - Box.bb2

BLITZ
; *** Open Blitz mode display
BitMap 0,320,256,5
Slice 0,44,320,256,$fff8,5,8,32,320,320
Show 0
; *** Alter palette
RGB 1,0,0,15
; *** Vertical boxes
For Y=0 To 256 Step 16
  ; *** Horizontal boxes
  For X=0 To 320 Step 16
    ; *** Toggle square colour
    COL=1-COL
    ; *** Draw square
    Box X,Y,X+15,Y+15,COL
  Next X
Next Y
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

BOXF

```
Mode(s):   Amiga/Blitz
Statement: draw a solid rectangle
Syntax:    Boxf X1,Y1,X2,Y2,COLOUR
```

BOXF is identical to the BOX statement except it is used to draw solid rectangular shapes, as opposed to outlines. X1 and Y1 are the coordinates of the top left-hand corner of the rectangle and X2 and Y2 are the coordinates of the bottom right-hand corner. COLOUR is the colour of the outline; a COLOUR parameter of (-1) will cause the rectangle to be inverted.

Simple but effective screen wipes can be created with BOXF. Here is an example:

```
; *** Screen wipe
; *** Filename - Wipe.bb2

BLITZ
; *** Open 2 BitMaps for double-buffering
BitMap 0,320,DispHeight,3
Cls 7
BitMap 1,320,DispHeight,3
Cls 7
; *** Define display Slice
Slice 0,44,320,256,$fff8,3,8,8,320,320
Show 0
; *** Starting co-ordinates for box
X1=160
X2=160
Y1=DispHeight/2
Y2=DispHeight/2
; *** Main loop
Repeat
  ; *** Draw rectangle
  Boxf X1,Y1,X2,Y2,0
  ; *** Decrease box size
  Let X1-1
  Let X2+1
  Let Y1+1
  Let Y2-1
  ; *** Wait for Vertical Blank
  VWait
  ; *** Double-buffering routine
  Show MAP : MAP=1-MAP : Use BitMap MAP
; *** Until co-ordinates meet
Until X1=0
; *** Return to Blitz Basic 2 editor
End
```

7.1.6 Circle circus

CIRCLE

```
Mode(s):   Amiga/Blitz
Statement: draw a circular or eliptical outline
Syntax:    Circle X,Y,RADIUS,COLOUR
Syntax 2:  Circle X,Y,RADIUS,YRADIUS,COLOUR
```

Drawing circles and elipses is very simple with Blitz Basic. Set the position of the centre of the circle using X and Y, followed by the radius of the circle.

If the optional YRADIUS parameter is included then an elipse may be drawn instead. COLOUR is the colour of the outline; a COLOUR parameter of (-1) will cause the circle to be inverted. The following example generates a dual-playfield circle effect, reminiscant of the "Spaceballs: State Of The Art" megademo:

```
; *** Demo circle effect
; *** Filename - Silly_Circles.bb2

BLITZ
; *** Open 2 BitMaps for double buffering
BitMap 0,640,512,3
BitMap 1,640,512,3
; *** Draw differently sized circles
For A=0 To 400 Step 10
  Circle 320,250,400-A,Rnd(7)+1
Next A
; *** Clone BitMap graphics
CopyBitMap 1,0
Slice 0,44,320,256,$fffa,6,8,16,640,640
Repeat
  VWait
  ; *** Define circular path
  X1=160+Sin(R)*160
  Y1=128+Cos(R)*128
  X2=160-Sin(R)*160
  Y2=128-Cos(R)*128
  ; *** Show foreground graphics
  ShowF 1,X1,Y1,X2
  ; *** Show background graphics
  ShowB 0,X2,Y2,X1
  Let R+0.05
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

CIRCLEF

```
bn:   Amiga/Blitz
Statement: draw a solid circle or elipse
Syntax:    Circlef X,Y,RADIUS,COLOUR
Syntax 2:  Circlef X,Y,RADIUS,YRADIUS,COLOUR
```

CIRCLEF works the same as CIRCLE except that it draws solid circles, as opposed to outlines.

If the optional YRADIUS parameter is included then an elipse may be drawn instead. COLOUR is the colour of the outline; a COLOUR parameter of (-1) will cause the circle to be inverted. For example:

```
; *** Solid Circles
; *** Filename - Circlef.bb2

BLITZ
; *** Open BLITZ mode display
BitMap 0,320,256,5
Slice 0,44,5
Show 0
; *** Draw 100 random circles and ellipses
For A=1 To 100
  Circlef Rnd(320),Rnd(256),Rnd(10)+5,Rnd(15)+2,Rnd(30)+1
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.1.7 Polygon power

POLY

```
Mode(s):   Amiga/Blitz
Statement: draw multiple line
Syntax:    Poly POINTS,COORDS.w,COLOUR
```

The POLY statement is another BitMap-based command which is used to draw multiple line objects. The COORDS.w parameter contains the co-ordinates of each point to join up, from either an array or NewType of words. In this way, complex outlines can be created using a single statement. COLOUR is the colour of the polygon. For example:

```
; *** Hyperspace
; *** Filename - Poly.bb2

NEWTYPE .HYP
  ; *** Define polygon co-ordinates
  XOFF.w
  YOFF
  X1
  Y1
End NEWTYPE
BLITZ
; *** Open 2 BitMaps for double buffering
BitMap 0,320,DispHeight,3
BitMap 1,320,DispHeight,3
Slice 0,44,3
Show 0
Mouse On
While Joyb(0)=0
  Cls
  ; *** Wait for Vertical Blank
  VWait
  ; *** Set polygon co-ordinates
  A.HYP\XOFF=Rnd(320),Rnd(256),MouseX,MouseY
  ; *** Draw polygon
  Poly 2,A,Rnd(7)+1
  ; *** Double-buffering routine
  Show MAP : MAP=1-MAP : Use BitMap MAP
Wend
; *** Return to Blitz Basic 2 editor
End
```

POLYF

```
Mode(s):   Amiga/Blitz
Statement: draw a solid polygon
Syntax:    Polyf POINTS,COORDS.w,COLOUR[,COLOUR2]
```

POLYF is used to draw polygons and is the filled equivalent of POLY. The COORDS.w parameter contains the co-ordinates of each point to join up, from either an array or NewType of words.

The optional COLOUR2 parameter, if included, will be used if the co-ordinates are listed in anti-clockwise order. If COLOUR2 is equal to (-1) then the polygon will not be drawn if the vertices are listed in anti-clockwise order. This is useful when designing three-dimensional objects to create depth. Here's an example:

```
; *** Polygon triangles
; *** Filname - Polyf.bb2

NEWTYPE .TRIG
  ; *** Define polygon co-ordinates
  XOFF.w
  YOFF
  X1
  Y1
  X2
  Y2
End NEWTYPE
BLITZ
; *** Open BLITZ mode display
BitMap 0,320,DispHeight,3
Slice 0,44,3
Show 0
; *** Repeat until mouse click
While Joyb(0)=0
  VWait
  ; *** Set polygon co-ordinates
  A.TRIG\XOFF=Rnd(320),Rnd(256),Rnd(320),Rnd(256),Rnd(320),Rnd(256)
  ; *** Draw polygon
  Polyf 3,A,Rnd(7)+1
Wend
; *** Return to Blitz Basic 2 editor
End
```

7.1.8 Fill her up!

FLOODFILL

```
Mode(s):   Amiga/Blitz
Statement: fill a screen region with colour
Syntax:    FloodFill X,Y,COLOUR[,BORDER]
```

The FLOODFILL statement will fill any part of the screen with a solid block of colour, starting at coordinates X,Y. If the optional BORDER parameter is included then the filled region will be surrounded by a border of that colour:

```
; *** Filling station
; *** Filename - FloodFill.bb2

BLITZ
; *** Open BLITZ mode display
BitMap 0,320,256,3
```

```
Slice 0,44,3
Show 0
; *** Fill screen ten times
For A=1 To 10
  FloodFill 1,1,A
Next A
; *** Return to Blitz Basic 2 editor
End
```

```
; *** FloodFill example 2
; *** Filename - FloodFill2.bb2

BLITZ
; *** Open BLITZ mode display
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
Box 1,1,319,199,1
Repeat
  ; *** Generate random colour
  COL=Int(Rnd(5)+2)
  ; *** Fill rectangle
  FloodFill 50,50,COL,1
  Locate 15,27
  ; *** Print current colour
  Colour COL
  NPrint "Colour ",COL
Until Joyb(0)>0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FREEFILL

```
Mode(s):   Amiga/Blitz
Statement: free 2D fill drawing memory
Syntax:    FreeFill
```

Blitz Basic uses a single monochrome BitMap the size of the BitMap being used to calculate its filled routines. FREEFILL will free any memory that Blitz uses to execute the commands BOXF, CIRCLEF and FLOODFILL. Only use FREEFILL if you don't need access to any of these commands. For example:

```
; *** FreeFill example
; *** Filename - FreeFill.bb2

BLITZ
; *** Open Blitz mode display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Fill screen ten times
For A=1 To 10
  FloodFill 1,1,A
Next A
; *** No more access to drawing commands!
FreeFill
; *** Illegal access!
For A=1 To 10
  FloodFill 1,1,A
Next A
; *** Wait for a mouse click
MouseWait
; *** End the show
End
```

7.2 Palettes

Palette objects, or palettes, are temporary storage areas of colour information. This information can be taken either from an IFF (Interchangeable File Format) file or created from scratch. If colour information is created by the user then it will not affect the current screen colours until the USE PALETTE statement has been executed.

7.2.1 Loading a palette object

LOADPALETTE

```
Mode(s):   Amiga
Statement: load a palette object
Syntax:    LoadPalette PALETTE#,"FILENAME"[,OFFSET]
```

LOADPALETTE loads a palette object from disk. The "FILENAME" parameter specifies the name of an IFF file (such as a DPaint picture) containing colour information. If the file contains colour cycling information, then this will also be loaded into the palette object. LOADPALETTE will not affect currently displayed colours until USE PALETTE is executed. For example:

```
; *** LoadPalette example
; *** Filename - LoadPalette.bb2

Screen 0,5,"Loading screen and palette"
F$="FILENAME.IFF"
; *** Load IFF screen from disk
LoadScreen 0,F$
; *** Load IFF screen's palette
LoadPalette 0,F$
; *** Add screen's palette to display
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.2.2 Controlling palette objects

USE PALETTE

```
Mode(s):   Amiga/Blitz
Statement: set current palette object
Syntax:    Use Palette PALETTE#
```

This statement sets the specified palette object as the current palette object. USE PALETTE is used to add the colours contained within a colour palette to the current display. Here is an example:

```
; *** Use Palette example
; *** Filename - Use Palette.bb2

For A=1 To 10
  ; *** Create custom palette
  PalRGB 0,A,Rnd(7),Rnd(7),Rnd(7)
Next A
; *** Open screen and grab its BitMap
Screen 0,3,"Colour screen"
ScreensBitMap 0,0
BitMapOutput 0
; *** Draw some BitMap graphics in default colours
For B=1 To 100
  Circlef Rnd(320),Rnd(200)+30,Rnd(10)+5,Rnd(15)+3,Rnd(6)+1
Next B
Locate 0,2
NPrint "Before Use Palette"
; *** Display custom palette
VWait 100
```

```
Use Palette 0
Locate 0,2
NPrint "After Use Palette "
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SHOWPALETTE

```
Mode(s):   Amiga/Blitz
Statement: display current palette object
Syntax:    ShowPalette PALETTE#
```

The SHOWPALETTE statement displays a palette object in the current screen or Slice. SHOWPALETTE must be executed after a palette object has been defined, otherwise it will not be visible. Here is an example:

```
; *** ShowPalette example
; *** Filename - ShowPalette.bb2

; *** Define a random palette
For A=0 To 10
  PalRGB 0,A,Rnd(9),Rnd(9),Rnd(9)
Next A
; *** Open screen and grab its BitMap
Screen 0,3,"ShowPalette"
ScreensBitMap 0,0
; *** Add palette to display
ShowPalette 0
; *** Plot a random starfield
For B=0 To 100
  Plot Rnd(320),Rnd(200)+20,Rnd(8)+1
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

NEWPALETTEMODE

```
Mode(s):   Amiga/Blitz
Statement: set output of Use Palette
Syntax:    NewPaletteMode On/Off
```

NEWPALETTEMODE is used to enhance compatibility with older Blitz Basic 2 programs. If NEWPALETTEMODE is set to (Off) then USE PALETTE will perform identically to SHOWPALETTE, and if it is set to (On) then USE PALETTE will perform as normal. This is because the USE PALETTE statement has been updated - and indeed superceeded - by the SHOWPALETTE statement.

FREE PALETTE

```
Mode(s):   Amiga/Blitz
Statement: erase a palette object
Syntax:    Free Palette PALETTE#
```

FREE PALETTE erases the contents of the palette object specified by PALETTE#. It does not affect the current display colours. Example:

```
; *** Free Palette example
; *** Filename - Free_Palette.bb2

; *** Define a random palette
For A=0 To 10
  PalRGB 0,A,Rnd(9),Rnd(9),Rnd(9)
Next A
; *** Open screen and grab its BitMap
Screen 0,3,"ShowPalette"
ScreensBitMap 0,0
; *** Add palette to display
ShowPalette 0
; *** Plot a random starfield
For B=0 To 100
  Plot Rnd(320),Rnd(200)+20,Rnd(8)+1
Next B
; *** Remove palette
Free Palette 0
Cls 0
; *** Plot another random starfield
For B=0 To 100
  Plot Rnd(320),Rnd(200)+20,Rnd(8)+1
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.2.3 Manipulating palettes

PALRGB

```
Mode(s):   Amiga/Blitz
Statement: set a colour register within a palette object
Syntax:    PalRGB PALETTE#,REGISTER,RED,GREEN,BLUE
```

The PALRGB statement allows you to set an individual colour register within a palette object. Values for REGISTER are taken from the Amiga's standard 32 colour registers. The colour change will not become evident until the USE PALETTE statement is used. Try the following example:

```
; *** PalRGB example
; *** Filename - PalRGB.bb2

; *** Define random colour palette
PalRGB 0,0,Rnd(7),Rnd(7),Rnd(7)
PalRGB 0,1,Rnd(15),Rnd(15),Rnd(15)
PalRGB 0,2,0,0,0
; *** Open screen (3 bitplanes)
Screen 0,3,"Colour screen"
; *** Add colour palette to display
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

RGB

```
Mode(s):   Amiga/Blitz
Statement: set a colour register to an RGB colour value
Syntax:    RGB REGISTER,RED,GREEN,BLUE
```

RGB allows you to set an individual colour register in a palette to an RGB colour value. Values for REGISTER are taken from the Amiga's standard 32 colour registers. RGB does not affect palette objects. For example:

```
; *** RGB example
; *** Filename - RGB.bb2

BLITZ
; *** Open BLITZ mode display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Change colour register 0, 15 times
For A=1 To 15
  RGB 0,Rnd(15),Rnd(15),Rnd(15)
  VWait 20
Next A
; *** Return to Blitz Basic 2 editor
End
```

The RED, GREEN and BLUE statements return the amount of their respected colour in a specified colour register. The returned values range from zero to 15.

RED

```
Mode(s):   Amiga/Blitz
Function:  return the amount of RGB red in a colour register
Syntax:    r=Red(REGISTER)
```

Values for REGISTER are taken from the Amiga's standard 32 colour registers. For example:

```
; *** Red example
; *** Filename - Red.bb2

; *** Open screen and grab its BitMap
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
; *** Set red to 8
RGB 0,8,0,0
Locate 0,3
; *** Returns "8"
NPrint "Red = ",Red(0)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

GREEN

```
Mode(s):   Amiga/Blitz
Function:  return the amount of RGB green in a colour register
Syntax:    g=Green(REGISTER)
```

Values for REGISTER are taken from the Amiga's standard 32 colour registers. For example:

```
; *** Green example
; *** Filename - Green.bb2

; *** Open screen and grab its BitMap
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
; *** Set green to 10
RGB 0,0,10,0
Locate 0,3
; *** Returns "10"
NPrint "Green = ",Green(0)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

BLUE

```
Mode(s):   Amiga/Blitz
Function:  return the amount of RGB blue in a colour register
Syntax:    b=Blue(REGISTER)
```

Values for REGISTER are taken from the Amiga's standard 32 colour registers. Here is an example:

```
; *** Blue example
; *** Filename - Blue.bb2

; *** Open screen and grab its BitMap
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
; *** Set blue to 14
RGB 0,0,0,14
Locate 0,3
; *** Returns "14"
NPrint "Blue = ",Blue(0)
```

```
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.3 Fades

The Blitz Basic fade commands can be used to create impressive screen wipes and transitions. Here's a brief overview...

7.3.1 Fading into and out of reality

FADEIN

```
Mode(s):   Blitz
Statement: fade in a colour palette
Syntax:    FadeIn PALETTE#[,RATE][,LOW,HIGH]
```

The FADEIN statement is used to fade in the palette of the current Slice from black, to the RGB values in PALETTE#. The optional RATE parameter allows you to control the speed of the fade (0 is the fastest fade). The optional LOW and HIGH parameters allow you to control which palette registers are affected by the fade. All palette registers between the values of LOW and HIGH will fade in. Consult the FADEOUT example.

FADEOUT

```
Mode(s):   Blitz
Statement: fade out a colour palette
Syntax:    FadeOut PALETTE#[,RATE][,LOW,HIGH]
```

The FADEOUT statement is used to fade out the palette of the current Slice from the RGB values in PALETTE#, to black. The optional RATE parameter allows you to control the speed of the fade (0 is the fastest fade). The optional LOW and HIGH parameters allow you to control which palette registers are affected by the fade. All palette registers between the values of LOW and HIGH will fade out.

Try the following example:

```
; *** Fading Example
; *** Filename - Fade.bb2

SPEED=2
BitMap 0,320,256,4
; *** Load IFF file to fade in
LoadBitMap 0,"FILENAME.IFF",0
BLITZ
Slice 0,44,4
```

```
; *** Set all colours to black
For A=0 To 15
  RGB A,0,0,0
Next A
Show 0
; *** Fade in image
VWait 100
FadeIn 0,SPEED
; *** Fade out image
VWait 100
FadeOut 0,SPEED
VWait 100
; *** Return to Blitz Basic 2 editor
End
```

7.3.2 Manual fading

If Blitz Basic's automatic fading isn't to your satisfaction then why not try the more powerful manual fading commands. These allow you much more control over the speed of the fade and enable you to synchronise screen fading with program execution.

ASYNCFADE

```
Mode(s):   Amiga/Blitz
Statement: control palette fading
Syntax:    ASyncFade On/Off
```

ASYNCFADE controls how the above fade commands operate. Normally, FADEIN and FADEOUT will halt program execution, fade, and then continue program execution (ASYNCFADE OFF - the default mode). ASYNCFADE ON switches this auto-fade off and DOFADE must be executed to perform the next step of the fade.

DOFADE

```
Mode(s):   Blitz
Statement: execute the next step of a fade
Syntax:    DoFade
```

The DOFADE statement executes the next step of a fade. It must be called after one of the above fade commands.

FADESTATUS

```
Mode(s):  Blitz
Function: return number of remaining fade steps
Syntax:   f=FadeStatus
```

FADESTATUS may be used in conjunction with the DOFADE statement to determine whether or not there are any more fade steps to execute. If a fade has finished then (0) is returned, and if there are fade steps left then (-1) is returned.

Here is a complete manual fade example:

```
; *** Manual fading
; *** Filename - DoFade.bb2

SPEED=2
BitMap 0,320,256,4
; *** Load IFF file to fade in
LoadBitMap 0,"FILENAME.IFF",0
BLITZ
Slice 0,44,4
; *** Set all colours to black
For A=0 To 15
  RGB A,0,0,0
Next A
; *** Turn manual fading on
ASyncFade On
Show 0
BitMapOutput 0
FadeIn 0,1
; *** Fade in picture
Repeat
  DoFade
  Let B+1
  Locate 0,0
  Print "Fade step ",B
  VWait 20
Until FadeStatus=0
; *** Return to Blitz Basic 2 editor
End
```

7.4 Colour cycling

If you are familiar with the Deluxe Paint series of programs then you will probably already know about the simplest form of colour cycling. This type makes each of the colours in the colour palette change places, or cycle.

SETCYCLE

```
Mode(s):   Amiga
Statement: define colour cycling for a specified palette
Syntax:    SetCycle PALETTE#,CYCLE#,LOW,HIGH[,SPEED]
```

The SETCYCLE statement is used to define colour cycling information for the CYCLE statement. PALETTE# is the number of the palette to cycle. You may define a maximum of seven different colour cycles for a single palette, each determined by a unique CYCLE# number. All palette registers between the values of LOW and HIGH will cycle. The optional SPEED parameter specifies the speed of the cycle, either (-1) or (1); a negative value will cycle the colours in reverse. For example:

```
; *** SetCycle example
; *** Filename - SetCycle.bb2

Screen 0,5
F$="FILENAME.IFF"
; *** Load IFF file and colour information
LoadScreen 0,F$
LoadPalette 0,F$
; *** Add colour palette to display
Use Palette 0
; *** Cycle backwards
SetCycle 0,0,1,10,-1
Cycle 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CYCLE

```
Mode(s):   Amiga
Statement: execute defined colour cycling
Syntax:    Cycle PALETTE#
```

CYCLE is used to execute the colour cycling information defined with SETCYCLE. PALETTE# is the number of the palette to cycle. Here is an example:

```
; *** A nice day for a Cycle
; *** Filename - Cycle.bb2

Screen 0,5
F$="FILENAME.IFF"
```

```
; *** Load IFF file and colour information
LoadScreen 0,F$
LoadPalette 0,F$
; *** Add colour palette to display
Use Palette 0
; *** Cycle palette
Cycle 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

STOPCYCLE

```
Mode(s):   Amiga
Statement: stop defined colour cycling
Syntax:    StopCycle
```

The STOPCYCLE statement stops all colour cycling in its tracks. For example:

```
; *** StopCycle example
; *** Filename - StopCycle.bb2

Screen 0,5
F$="FILENAME.IFF"
; *** Load IFF screen and colour information
LoadScreen 0,F$
LoadPalette 0,F$
; *** Add colour palette to display
Use Palette 0
; *** Cycle colour palette
Cycle 0
MouseWait
; *** Stop colour palette cycling
StopCycle
VWait 100
; *** Return to Blitz Basic 2 editor
End
```

7.5 Copper Lists

The Amiga's co-processor, or Copper, is used to generate subtly coloured "rainbow" backgrounds, and to create special display effects. Because the Copper works in parallel it can execute instructions at the same time as the main processor.

7.5.1 Copper load of this

COLSPLIT

```
Mode(s):   Amiga/Blitz
Statement: control palette colour registers
Syntax:    ColSplit REGISTER,RED,GREEN,BLUE,Y
```

If you've ever marvelled at the colourful "rainbows" that seem to be part of every platform game or demo, and wondered how to create them in Blitz Basic then look no further.

The COLSPLIT statement is used to change the palette colour registers at a position relative to the top of the current Slice. As will be explained in the next chapter, the Amiga's hardware provides 32 colour registers. However, only colour registers zero through 15 can be affected by COLSPLIT. In practice, this colour can be assigned a different value for each horizontal scan line, to create really smooth colour graduations.

RED, GREEN and BLUE are the RGB components of the colour register, and the Y parameter specifies a vertical offset from the top of the Slice. Here are some examples:

```
; *** Simple copper - planet mars
; *** Filename - ColSplit1.bb2

BLITZ
; *** Open BLITZ mode display (1 bitplane)
BitMap 0,320,260,1
Slice 0,44,320,260,$fff8,1,8,2,320,320
Show 0
; *** Define colour registers 1 through 11
For A=1 To 11
  ColSplit 0,A,0,A,A*20
Next
; *** Define colour register 0
ColSplit 0,0,0,0,260
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

The second example gives a useful demonstration of how the copper instructions can be used to generate "rainbow text". This is where a copper list is placed behind a text string to create multi-colour text. To produce rainbow text, the text must be generated using the same colour register as is affected by COLSPLIT (the copper list must also be placed at the same y co-ordinate as the text!):

```
; *** ColSplit example 2
; *** Filename - ColSplit2.bb2

BLITZ
; *** Open BLITZ mode display (1 bitplane)
BitMap 0,320,256,1
Slice 0,44,320,256,$fff8,1,8,2,320,320
Show 0
BitMapOutput 0
For A=0 To 7
  ColSplit 0,A,A,A,A
Next A
; *** Generate upper rainbow
For B=8 To 15
  ColSplit 0,15-B,15-B,15-B,B
Next B
For C=0 To 7
  ColSplit 0,C,C,C,55+C
Next C
; *** Generate lower rainbow
For D=8 To 15
  ColSplit 0,15-D,15-D,15-D,55+D
Next D
For E=0 To 7
  ColSplit 1,E,0,E,20+E*3
Next E
; *** Text message to display between rainbows
Locate 1,4
Print "Cool copper bars Blitz Basic 2 style!"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.5.2 Custom copper lists

If the COLSPLIT statement is not powerful enough for your needs then why not take a look at the other Copper-based statement, CUSTOMCOP. This allows the advanced Blitz Basic programmer to introduce custom copper instructions.

CUSTOMCOP

```
Mode(s):   Amiga/Blitz
Statement: create custom copper lists
Syntax:    CustomCop SOURCE$,Y
```

The CUSTOMCOP statement is used to execute custom copper instructions at a specified position from the top of the display. SOURCE$ is a string of characters equivalent to a series of copper instructions. The Y parameter is the y offset of the copper list. Custom copper lists are not for the faint hearted! Try the following example:

```
; *** CustomCop example
; *** Filename - CustomCop.bb2

BLITZ
; *** Some hardware trickery
#BPLMOD1=$108
#BPLMOD2=$10A
; *** Open BLITZ mode display
BitMap 0,640,256,3
Slice 0,44,320,256,$FFF8,3,8,32,640,640
Show 0

; *** Create mountain landscape
RGB 2,9,0,0
Y=100 : LAND=3 : DI=-1
For X=0 To 640
  D=Int(Rnd(LAND))
  If D=1 Then DI=-1
  If D=2 Then DI=1
  Let Y+DI
  If Y<0 Then Y=0
  If Y>356-1 Then Y=355
  Line X,356,X,Y,2
Next X

; *** Mirror mountain
ColSplit 2,0,0,8,150
CO$=Mki$(#BPLMOD1)+Mki$(-122)
CO$+Mki$(#BPLMOD2)+Mki$(-122)
CustomCop CO$,150+44

; *** Scroll display
For X=0 To 320
  VWait
Show 0,X,0
Next X
; *** Return to Blitz Basic 2 editor
End
```

7.5.3 Copper list functions

The following functions are used to obtain information about the Blitz mode copper list.

COPLOC

```
Mode(s):  Amiga/Blitz
Function: return the memory address of Blitz mode copper list
Syntax:   c=CopLoc
```

Blitz Basic merges all Slices and copper lists into a single copper list. The COPLOC function returns the memory address of the Blitz mode copper list. For example:

```
; *** CopLoc example
; *** Filename - CopLoc.bb2

BLITZ
; *** Open BLITZ mode display
BitMap 0,320,260,1
Slice 0,44,320,260,$fff8,1,8,2,320,320
Show 0
BitMapOutput 0
; *** Create a simple copper list
For A=1 To 10
  ColSplit 1,A,A,A,A*2
Next
; *** Output address of copper list
Locate 0,1
NPrint CopLoc
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

COPLEN

```
Mode(s):  Amiga/Blitz
Function: return the length of Blitz mode copper list
Syntax:   c=CopLen
```

Blitz Basic merges all Slices and copper lists into a single copper list. COPLEN returns the length, in bytes, of the Blitz mode copper list. Try the following example:

```
; *** CopLen example
; *** Filename - CopLen.bb2

BLITZ
; *** Open BLITZ mode display
BitMap 0,320,260,1
Slice 0,44,320,260,$fff8,1,8,2,320,320
Show 0
BitMapOutput 0
; *** Output length, in bytes, of copper list
NPrint CopLen," bytes before rainbow."
VWait 100
Cls
; *** Create simple copper list
For A=1 To 11
  ColSplit 0,0,A,A,A*20
Next
ColSplit 0,0,0,0,260
; *** Output new length of copper list
Locate 0,0
NPrint CopLen," bytes after rainbow."
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

7.6 IFF Animation

Blitz Basic can also display full-screen IFF animations, such as those created with the suberb Deluxe Paint IV program. Remember that, the larger the animation, and the more colours involved, the more memory intensive the animation will be! Animations with fewer colours do tend to run faster when displayed using the powerful Blitz Basic animation commands. Here's how...

7.6.1 Animated antics

LOADANIM

```
Mode(s):   Amiga
Statement: load an IFF animation into memory
Syntax:    LoadAnim ANIM#,"FILENAME"[,PALETTE#]
```

The LOADANIM statement is used to load an IFF animation into memory. In order to create the correct screen size and resolution for the animation you may use the ILBMINFO statement. The optional PALETTE# parameter can be used to load the animation's colour palette into memory. Try the following example:

```
; *** Loading Animations
; *** Filename - LoadAnim.bb2

F$="FILENAME.ANIM"
; *** Analyse animation
ILBMInfo F$
; *** Open screen to animation dimensions
Screen 0,0,0,ILBMWidth,ILBMHeight,ILBMDepth,ILBMViewMode,"",1,2
; *** Grab screen's BitMap
ScreensBitmap 0,0
Bitmap 1,ILBMWidth,ILBMHeight,ILBMDepth
; *** Load animation
LoadAnim 0,F$,0
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

INITANIM

```
Mode(s):   Amiga/Blitz
Statement: initialise animation
Syntax:    InitAnim ANIM#[,BITMAP#]
```

INITANIM renders the first frame of the animation onto the current BitMap and, if the optional BITMAP# parameter is included, renders the second frame onto the specified BitMap. This is for creating flicker-free double-buffered animations. For example:

```
; *** InitAnim example
; *** Filename - InitAnim.bb2

F$="FILENAME.ANIM"
; *** Analyse animation
ILBMInfo F$
; *** Open screen to animation dimensions
Screen 0,0,0,ILBMWidth,ILBMHeight,ILBMDepth,ILBMViewMode,"",1,2
; *** Grab screen's BitMap
ScreensBitMap 0,0
BitMap 1,ILBMWidth,ILBMHeight,ILBMDepth
; *** Load animation
LoadAnim 0,F$,0
Use Palette 0
; *** Render 1st and 2nd frames of animation
InitAnim 0,0
; *** Wait for a mouse click
```

```
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

NEXTFRAME

```
Mode(s):   Amiga/Blitz
Statement: draw next animation frame
Syntax:    NextFrame ANIM#
```

Rendering of frames to the current BitMap is achieved through the use of the NEXTFRAME statement. If the last frame of the animation has been rendered NEXTFRAME will automatically loop back to the beginning of the animation. Here is an example:

```
; *** NextFrame example
; *** Filename - NextFrame.bb2

; *** Repeat until mouse click
While Joyb(0)=0
  ; *** Double buffering routine
  ShowBitMap DB
  VWait SPEED
  DB=1-DB
  Use BitMap DB
  ; *** Loop back to beginning of animation
  NextFrame 0
Wend
; *** Return to Blitz Basic 2 editor
End
```

FRAMES

```
Mode(s):  Amiga/Blitz
Function: return the number of frames in an animation
Syntax:   f=Frames(ANIMATION)
```

The FRAMES function simply returns the number of frames in a specified animation. This is useful if, for example, you want to stop an animation before it loops. For example (load an animation into memory prior to the following code):

```
; *** Frames example
; *** Filename - Frames.bb2

F=Frames(0)
```

```
A=1
; *** Repeat until last frame is reached
While A<=F
  ; *** Double buffering routine
  ShowBitmap DB
  VWait
  DB=1-DB
  Use Bitmap DB
  ; *** Next frame of animation
  NextFrame 0
  Let A+1
Wend
; *** Return to Blitz Basic 2 editor
End
```

7.6.2 A full example

The following example is a general-purpose animation viewer. It uses the double-buffering technique explained above to create perfect, flicker-free animations. Remember to insert your own filename into F$. Example:

```
; *** Displaying an animation
; *** Filename - Animation_example.bb2

F$="FILENAME.ANIM"
SPEED=1 ; *** Frame delay
; *** Analyse animation
ILBMInfo F$
; *** Open screen to animation dimensions
Screen 0,0,0,ILBMWidth,ILBMHeight,ILBMDepth,ILBMViewMode,"",1,2
; *** Grab screen's BitMap
ScreensBitMap 0,0
BitMap 1,ILBMWidth,ILBMHeight,ILBMDepth
; *** Load animation
LoadAnim 0,F$,0
Use Palette 0
; *** Initialise animation
InitAnim 0,0
While Joyb(0)=0
  ; *** Double buffering routine
  ShowBitMap DB
  VWait SPEED
  DB=1-DB
  Use BitMap DB
  ; *** Next frame of animation
  NextFrame 0
Wend
; *** Return to Blitz Basic 2 editor
End
```

7.7 End-of-Chapter summary

Pixels are the thousands of tiny elements which make up the Amiga's display. Single pixels are plotted using the PLOT statement. Pixel colours are read using the POINT statement.

Table 7.1 : 2D drawing commands

Shape	Command
Square	BOX/BOXF
Rectangle	BOX/BOXF
Circle	CIRCLE/CIRCLEF
Ellipse	CIRCLE/CIRCLEF
Polygon	POLY/POLYF

Palettes are temporary storage areas of colour information. This information can be taken either from an IFF (Interchangeable File Format) file or created from scratch using PALRGB or RGB.

The COLSPLIT statement is used manipulate the Copper chip in order to create colour rainbows. Custom copper lists can be created with the CUSTOMCOP statement. The Copper can be programmed to mirror text, stretch graphics and clone images. For a full example of custom copper lists please consult Appendix 2.

Blitz Basic can display and manipulate standard Deluxe Paint animations. Remember that, the larger the animation, and the more colours involved, the more memory intensive the animation will be! Animations with fewer colours do tend to run faster when displayed in Blitz.

Chapter 8 : Sprites and Shapes

This chapter covers the manipulation of sprites and shape objects in Blitz Basic 2. It will also show you the finer points of collision detection. Here goes...

8.1 Sprites

What is a sprite? Well, it's an object which can move across the screen - a monster or car - independently of other objects or the background. Sprites are initialised by either loading them from disk, or by converting a shape object into a sprite object using the GETASPRITE statement.

Sprites are handled entirely by the Amiga's hardware so they do not interfere or corrupt BitMap graphics in any way. Basically this means that sprites do not have to be erased manually when they are moved. However, there are some limitations that must be observed when using sprites:

- Sprites are only available in Blitz mode
- Sprites must be of either three or 15 colours (two/four bitplanes)

The resolution of all sprites corresponds to the lowest screen resolution (i.e. 320*200 or 320*256 pixels). Sprite co-ordinates are also always given in the lowest resolution (320*200 or 320*256).

Slices can display a maximum of eight sprites. This is because sprites are displayed by the Amiga's eight sprite channels, numbered (0) through (7).

If you are displaying a three-colour sprite, you may specify any of the eight sprite channels (0 through 7). However, if you are displaying a 15-colour sprite, you may only specify an even-numbered sprite channel (e.g. 0,2,4 or 6). Because 15-colour sprites require two sprite channels, they also need to use the associated odd-numbered sprite channel. For example, displaying a 15-colour sprite through sprite channel (2) will make sprite channel (3) unavailable.

The Amiga's hardware limits individual sprites to a maximum width of 16 low-resolution pixels. All sprites are therefore 16 pixels wide and have selectable height. However, Blitz Basic allows you to display sprites of greater width by splitting a shape up into groups of sixteen pixels. This means that a sprite may take up more than one sprite channel.

The number of sprite channels needed can be worked out using the following formulae:

```
For 3-colour sprites use :  CHANNELS=(WIDTH/16)
For 15-colour sprites use : CHANNELS=(WIDTH/16) * 2
```

For example, a 32 pixel wide 3-colour sprite displayed through sprite channel (2) will actually be converted to two 16 pixel wide sprites displayed through channels (2) and (3) - (32 pixels wide/16 = two sprite channels).

Similarly, a 48 pixel wide 15-colour sprite displayed through sprite channel (0) will take up sprite channels (0) through (5) - ((48 pixels wide/16) * 2 = six sprite channels!).

All sprite colours are taken from the Amiga's standard 32 colour registers, but the number of registers needed depend on the number of colours and the sprite channels involved.

Fifteen-colour sprites take their RGB values from colour registers 17 through 31. These are initially taken from the current Slice palette, but can be altered using the RGB statement. This means that, to display a 15-colour sprite on a 32 colour Slice, you would create your background or palette in 32-colour mode, and draw your sprites using colour numbers 17 to 31 only. When you come to display your sprites they will be drawn the correct colour.

Three-colour sprites, however, take on RGB values depending upon the sprite channels being used to display them. Each pair of three-colour sprite channels (0/1,2/3,4/5 and 6/7) use the same colour registers for definition of sprite colours. The following table shows the colour register assignment:

Table 8.1 : Sprite colour registers

Sprite channel	Transparent	Colour registers
0,1	16	17-19
2,3	20	21-23
4,5	24	25-27
6,7	28	29-31

Note that for each pair of sprites there is one register that is transparent, and three colour registers. So, to display a three-colour sprite on a 32 colour Slice, you would draw your sprites using colour numbers 17 to 19 only.

8.1.1 Loading sprites from disk

LOADSPRITES

```
Mode(s):   Amiga
Statement: load a range of sprites from disk
Syntax:    LoadSprites FIRST[,LAST],"FILENAME"
```

The LOADSPRITES statement is used to load a range of sprites into memory from disk. The FIRST parameter is the number of the first sprite to load from a previously created sprite bank, and the optional LAST parameter specifies the number of the last sprite to load. Try the following example:

```
; *** LoadSprites example
; *** Filename - LoadSprites.bb2

LoadSprites 0,"SPRITES"
BitMap 0,320,256,4
BLITZ
Slice 0,44,4
Show 0
ShowSprite 0,150,100,0
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

8.1.2 Saving sprites to disk

SAVESPRITES

```
Mode(s):   Amiga
Statement: save a range of sprites to disk
Syntax:    SaveSprites FIRST,LAST,"FILENAME"
```

This statement is used to save a range of sprites to disk from memory. The FIRST parameter is the number of the first sprite to save from a sprite bank held in memory, and the LAST parameter specifies the number of the last sprite to save

```
; *** SaveSprites example
; *** Filename - SaveSprites.bb2

BitMap 0,320,256,4
Boxf 0,0,20,20,3
GetaShape 0,0,0,20,20
GetaSprite 0,0
SaveSprites 0,0,"RAM:SPRITE"
Cls
Free Sprite 0
LoadSprites 0,"RAM:SPRITE"
BLITZ
Slice 0,44,4
Show 0
ShowSprite 0,150,100,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.1.3 Sprite commands

SHOWSPRITE

```
Mode(s):   Amiga/Blitz
Statement: display a Sprite on the screen
Syntax:    ShowSprite SPRITE#,X,Y,CHANNEL
```

This statement puts a hardware sprite on the screen, whose resolution corresponds to the current screen resolution. The X and Y parameters specify the coordinates of the sprite (in low-resolution pixels only). The Amiga hardware sprites can be controlled using channel numbers 0-7. Here is an example:

```
; *** ShowSprite example
; *** Filename - ShowSprite.bb2

BitMap 0,320,256,4
Circle 10,10,9,3
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
BLITZ
Slice 0,44,4
Show 0
For X=0 To 320
  VWait
  ShowSprite 0,X,20,0
Next X
; *** Return to Blitz Basic 2 editor
End
```

GETASPRITE

```
Mode(s):   Amiga/Blitz
Statement: convert shape object to a sprite object
Syntax:    GetaSprite SPRITE#,SHAPE#
```

The GETASPRITE statement converts a shape object to a sprite object. SHAPE# is the number of a previously initialised shape object to convert and SPRITE# is the number of the destination sprite object. For example:

```
; *** GetaSprite example
; *** Filename - GetaSprite.bb2

BitMap 0,320,256,2
Boxf 0,0,63,63,2
GetaShape 0,0,0,32,32
GetaSprite 0,0
Free Shape 0
Cls
BLITZ
Slice 0,44,2
Show 0
For A=0 To 3
  RGB A*4+17,15,15,0
  RGB A*4+18,15,8,0
  RGB A*4+19,15,4,0
Next A
```

```
For X=0 To 320
  VWait
  ShowSprite 0,X,20,0
Next X
; *** Return to Blitz Basic 2 editor
End
```

INFRONT

```
Mode(s):   Amiga/Blitz
Statement: convert sprite display to infront/behind a BitMap
Syntax:    InFront CHANNEL
```

One of the great features of hardware sprites is that they may be displayed in front of or behind any BitMap graphics. The INFRONT statement is used to convert sprite display to infront/behind BitMaps. CHANNEL must be an even number of value 0, 2, 4, 6 or 8. Sprites displayed using sprite channels greater than or equal to CHANNEL will appear behind any BitMap graphics, whilst those less than CHANNEL will appear in front:

```
; *** Using InFront
; *** Filename - InFront.bb2

; *** ... Create sprite here

InFront 4
; *** ... Insert display routines

; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End

In the above example (pseudo code only), sprites 4, 5, 6 and 7 will appear
behind and sprites 0, 1, 2 and 3 will appear in front. Here is a full
example:

; *** InFront example
; *** Filename - InFront2.bb2

PalRGB 0,1,15,0,15
BitMap 0,320,256,2
Boxf 0,0,63,63,1
GetaShape 0,0,0,64,64
GetaSprite 0,0
Cls
BitMapOutput 0
Locate 10,10
For WORDS=1 To 50
```

```
    Locate Rnd(30)+5,Rnd(20)+3
    Print "Hello"
  Next WORDS
  BLITZ
  Slice 0,44,2
  Show 0
  Use Palette 0
  For A=0 To 3
    RGB A*4+17,15,15,0
    RGB A*4+18,15,8,0
    RGB A*4+19,15,4,0
  Next A
  InFront 4
  For X=0 To 320
    VWait
    ShowSprite 0,X,20,0
    ShowSprite 0,X,120,4
  Next X
  ; *** Return to Blitz Basic 2 editor
  End
```

Note that you should only use the INFRONT statement with single-playfield Slices. If you want to create some dual-playfield Slice magic then use the following two commands.

INFRONTF

```
Mode(s):   Amiga/Blitz
Statement: dual playfield version of InFront (foreground)
Syntax:    InFrontF CHANNEL
```

The INFRONTF statement is used with dual-playfield Slices to control sprite/playfield priority with respect to the foreground playfield. CHANNEL must be an even number of value 0,2,4,6 or 8. Sprites displayed using sprite channels greater than or equal to CHANNEL will appear behind any BitMap graphics, whilst those less than CHANNEL will appear in front.

INFRONTB

```
Mode(s):   Amiga/Blitz
Statement: dual playfield version of InFront (background)
Syntax:    InFrontB CHANNEL
```

The INFRONTB statement is used with dual-playfield Slices to control sprite/playfield priority with respect to the background playfield. CHANNEL must be an even number of value 0,2,4,6 or 8. Sprites displayed using sprite channels greater than or equal to CHANNEL will appear behind any BitMap graphics, whilst those less than CHANNEL will appear in front:

```
; *** InFrontF/InFrontB example
; *** Filename - InFrontB.bb2

BitMap 1,320,256,2
Boxf 80,50,240,150,3
BitMap 0,320,256,2
Boxf 0,0,63,63,1
GetaShape 0,0,0,32,32
GetaSprite 0,0
Free Shape 0
Cls
Circlef 160,100,90,3
Circlef 160,100,50,0
BLITZ
Slice 0,44,320,256,$fff2,4,8,32,320,320
ShowF 0
ShowB 0,10,0
For A=0 To 3
  RGB A*4+17,15,15,0
  RGB A*4+18,15,8,0
  RGB A*4+19,15,4,0
Next A
InFrontF 4
InFrontF 2
InFrontB 4
For X=0 To 320
  VWait
  ShowSprite 0,X,20,0
  ShowSprite 0,X,80,2
  ShowSprite 0,X,140,4
Next X
; *** Return to Blitz Basic 2 editor
End
```

8.2 Shapes

The Amiga range of computers have access to an extremely powerful graphic shifter called the Blitter chip. Blitter Objects, or "Bobs" for short, are images which can be displayed on screen with lightning speed, but must be displayed and updated by the user to avoid graphic corruption. For reasons know only to Acid Software, Blitz Basic refers to these Bobs as shapes, or shape objects. These shape objects may be used in a variety of different ways, such as gadgets, menu items or game graphics.

Many commands are available for the purpose of drawing shapes onto a BitMap. These commands use the Amiga's blitter chip to achieve this, and are therefore very fast. The process of putting a shape onto a BitMap using the blitter is often referred to as "blitting" a shape.

The blitting speed of a shape is affected by its size and the blitting technique (in Blitz Basic there are three main blitting techniques). Obviously, larger shapes take longer to "blit" than smaller ones. Also, shapes with more colours take longer to blit.

The technique used to draw a shape also affects its speed. The fastest blitting command is the BLIT statement, however this provides no way of erasing the shape to allow for movement. QBLIT allows for movement, but it does corrupt BitMap graphics in the process. The most powerful blitting command, BBLIT, is also the slowest, as it allows for movement and doesn't corrupt any BitMap graphics.

8.2.1 Loading and saving shapes

LOADSHAPE

```
Mode(s):   Amiga
Statement: load an IFF file into a shape object
Syntax:    LoadShape SHAPE#,"FILENAME"[,PALETTE#]
```

This statement loads an IFF file (such as a DPaint picture) into a shape object. The optional PALETTE# parameter is used to load the colour information contained in the file into a palette object. Here is an example:

```
; *** LoadShape example
; *** Filename - LoadShape.bb2

Screen 0,3
ScreensBitMap 0,0
LoadShape 0,"A SHAPE.IFF",0
Use Palette 0
Blit 0,30,30
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SAVESHAPE

```
Mode(s):   Amiga
Statement: save a shape object to an IFF file
Syntax:    SaveShape SHAPE#,"FILENAME"[,PALETTE#]
```

SAVESHAPE saves the information contained in a shape object into an IFF file. The optional PALETTE# parameter allows you to save the shape's colour information as well. For example:

```
; *** SaveShape example
; *** Filename - SaveShape.bb2

BLITZ
BitMap 0,320,256,5
Slice 0,44,5
```

```
Show 0
BitMapOutput 0
For A=1 To 50
  Locate Rnd(30),Rnd(20)
  Colour Rnd(30)+1,Rnd(30)+1
  NPrint "Totally flipped!"
Next A
GetaShape 0,0,0,320,200
QAMIGA
SaveShape 0,"RAM:SHAPE.IFF"
Cls
LoadShape 0,"RAM:SHAPE.IFF"
Blit 0,0,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LOADSHAPES

```
Mode(s):   Amiga
Statement: load a range of shapes from disk
Syntax:    LoadShapes FIRST,[,LAST],"FILENAME"
```

The LOADSHAPES statement is used to load a range of shapes into memory from disk. The FIRST parameter specifies the number of the first shape object to be loaded. If the optional LAST parameter is included then only the shapes up to and including this value will be loaded:

```
; *** LoadShapes example
; *** Filename - LoadShapes.bb2

LoadShapes 0,"SHAPES"
BLITZ
BitMap 0,320,256,5
Slice 0,44,5
Show 0
Use Palette 0
; *** Blit first shape in range
Blit 0,0,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SAVESHAPES

```
Mode(s):   Amiga
Statement: save a range of shapes to disk
Syntax:    SaveShapes FIRST,LAST,"FILENAME"
```

SAVESHAPES is used to save a range of shapes to disk. The FIRST parameter specifies the number of the first shape object to be saved, and the LAST parameter specifies the number of the last shape object to be saved. Here's an example:

```
; *** SaveShapes example
; *** Filename - SaveShapes.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Boxf 0,0,10,10,2
GetaShape 0,0,0,10,10
Boxf 0,0,10,10,3
GetaShape 1,0,0,10,10
QAMIGA
SaveShapes 0,2,"RAM:SHAPES"
Cls
LoadShapes 0,"RAM:SHAPES"
Blit 0,140,100
Blit 1,160,100
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.2.2 Grabbing shapes

GETASHAPE

```
Mode(s):   Amiga/Blitz
Statement: grab a BitMap image into a shape object
Syntax:    GetaShape SHAPE#,X,Y,WIDTH,HEIGHT
```

Grabbing chunks of BitMaps is a speciality of Blitz Basic. The GETASHAPE statement copies a rectangular area of the currently used BitMap into the shape object specified by SHAPE#. The X and Y parameters are the coordinates of the top left of the box and the WIDTH and HEIGHT parameters specify the size of the area in pixels. Try the following example:

```
; *** GetaShape example
; *** Filename - GetaShape.bb2

Screen 0,3,"My Blobs"
ScreensBitMap 0,0
Circlef 100,100,10,5
GetaShape 0,80,80,120,120
For A=1 To 10
  Blit 0,Rnd(100)+10,Rnd(100)+30
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.2.3 Manipulating shapes

COPYSHAPE

```
Mode(s):   Amiga/Blitz
Statement: copy one shape object to another shape object
Syntax:    CopyShape SOURCE,DESTINATION
```

The COPYSHAPE statement copies one shape object (SOURCE) into another shape object (DESTINATION). This is a quick and simple way of creating "carbon copies" of shapes. For example:

```
; *** CopyShape example
; *** Filename - CopyShape.bb2

PalRGB 0,1,15,15,15
Screen 0,3,"Hello"
ScreensBitMap 0,0
GetaShape 0,0,0,50,10
For A=1 To 4
  CopyShape 0,A
  Blit A,50,50+(A*20)
Next A
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

HANDLE

```
Mode(s):   Amiga/Blitz
Statement: set reference point for all shape coordinate calculations
Syntax:    Handle SHAPE#,X,Y
```

The HANDLE statement sets the reference point of the shape object, SHAPE#. The handle offset (X,Y) is measured in pixels from the top left-hand corner of the shape. For example:

```
; *** Handle example
; *** Filename - Handle.bb2

Screen 0,3,"My Blobs"
ScreensBitMap 0,0
BitMapOutput 0
Circle 100,100,10,6
GetaShape 0,80,80,120,120
Cls
NPrint "Default handle"
Blit 0,50,50
Locate 0,6
NPrint "User handle"
Handle 0,40,40
Blit 0,50,50
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

For those who don't fancy getting their hands dirty, Blitz Basic provides an automatic method of centring these reference points.

MIDHANDLE

```
Mode(s):   Amiga/Blitz
Statement: set reference point to shape's centre
Syntax:    MidHandle SHAPE#
```

Here is an example:

```
; *** MidHandle example
; *** Filename - MidHandle.bb2

Screen 0,3
ScreensBitMap 0,0
```

```
BitMapOutput 0
Circle 100,100,10,6
GetaShape 0,80,80,120,120
Cls
NPrint "Using Midhandle"
MidHandle 0
Blit 0,70,60
VWait 100
Cls
Locate 0,0
NPrint "Manual handle"
Handle 0,ShapeWidth(0)/2,ShapeHeight(0)/2
Blit 0,70,60
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.2.4 Shape functions

SHAPEWIDTH

```
Mode(s):  Amiga/Blitz
Function: return the width of a shape object
Syntax:   w=ShapeWidth SHAPE#
```

SHAPEWIDTH returns the width of a shape object in pixels.

SHAPEHEIGHT

```
Mode(s):  Amiga/Blitz
Function: return the height of a shape object
Syntax:   h=ShapeHeight SHAPE#
```

SHAPEHEIGHT returns the height of a shape object in pixels. For example:

```
; *** Shape dimensions example
; *** Filename - ShapeHeight.bb2

VWait 20
BLITZ
BitMap 0,320,256,3
For A=0 To 10
Boxf 10,10,200,200,Rnd(6)+1
  GetaShape A,10,10,Rnd(100)+60,Rnd(100)+70
Next A
Cls
```

```
BitMapOutput 0
Slice 0,44,3
Show 0
For B=0 To 10
  Blit B,10,50
  W=ShapeWidth(B)
  H=ShapeHeight(B)
  Locate 0,0
  NPrint "Shape  : ",B
  NPrint "Width  : ",W," pixels"
  NPrint "Height : ",H," pixels"
  VWait 50
  Cls
Next B
; *** Return to Blitz Basic 2 editor
End
```

8.2.5 Automatic shape flipping

The characters in most commercial computer games are composed of hundreds of individual frames of animation. There are frames where the object is animating left to right, and up and down. Naturally, the more frames you have, the greater the disk space needed to store the objects, and the greater the amount of memory needed to display the objects.

Instead of storing seperate objects in the shape bank for reversed images, you can use Blitz 2's automatic shape flipping commands.

XFLIP

```
Mode(s):   Amiga/Blitz
Statement: flip a shape horizontally
Syntax:    XFlip SHAPE#
```

This statement reverses a shape object horizontally about the y-axis, thus creating a mirror image. XFLIP replaces the old shape object with this mirror image. To avoid this happening, use the COPYSHAPE statement to "clone" a shape and perform all shape manipulation commands on this instead. Try the following example:

```
; *** XFlip example
; *** Filename - XFlip.bb2

PalRGB 0,1,15,15,15
BLITZ
BitMap 0,320,256,1
BitMapOutput 0
Locate 1,1
NPrint "Totally flipped!"
Box 7,5,134,17,1
```

```
GetaShape 0,7,5,134,17
Cls
CopyShape 0,1
XFlip 1
Slice 0,44,1
Use Palette 0
Show 0
Blit 0,10,10
Blit 1,4,25
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

To mirror a shape object about the x-axis (i.e. vertically) you use the following statement.

YFLIP

```
Mode(s):   Amiga/Blitz
Statement: flip a shape vertically
Syntax:    YFlip SHAPE#
```

For example:

```
; *** YFlip example
; *** Filename - YFlip.bb2

BLITZ
BitMap 0,320,256,5
BitMapOutput 0
For A=1 To 50
  Locate Rnd(30),Rnd(20)
  Colour Rnd(30)+1,Rnd(30)+1
  NPrint "Totally flipped!"
Next A
GetaShape 0,0,0,320,100
Cls
CopyShape 0,1
YFlip 1
Slice 0,44,5
Use Palette 0
Show 0
Blit 0,0,0
Blit 1,0,105
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.2.6 Shape scaling and rotation

If you are familiar with the Super Nintendo console then you will be aware of its special graphics mode: Mode 7. Mode 7 uses the SNES hardware to bend, rotate, twist, and scale graphics to create totally new images - and in realtime too!. Whilst Blitz Basic is no match for the Super Nintendo's custom hardware, it can be used to create some wonderful special effects, such as scaling and rotation.

SCALE

```
Mode(s):   Amiga/Blitz
Statement: scale a shape object
Syntax:    Scale SHAPE#,X_RATIO,Y_RATIO[,PALETTE#]
```

Blitz Basic allows direct manipulation of a shape's size, although unfortunately not in realtime. The SCALE statement is used to stretch and shrink shape objects beyond recognition!

The X_RATIO and Y_RATIO parameters control the ratio of the enlargement/reduction. They are fully independent of each other and as such, different scaling can be applied to each axis:

Table 8.2 : SCALE ratios

```
RATIO  Effect
===============================
<1     Reduction in size
=1     No reduction/enlargement
>1     Enlargement in size
```

The optional PALETTE# parameter is used to specify a palette object for use in the scaling operation. If a palette is supplied then a shape may be shrunk without experiencing a loss in detail.

Try the following example, which uses the SCALE statement to magnify a text string. The routine takes a little while to generate the text, so do be patient!:

```
; *** Shape Scaling
; *** Filename - Scale.bb2

PalRGB 0,1,15,15,15
BLITZ
BitMap 0,320,256,1
BitMapOutput 0
Slice 0,44,1
Use Palette 0
Show 0
Locate 1,1
NPrint "Magnified text!!"
GetaShape 0,7,5,134,17
Cls
CopyShape 0,1
```

```
Scale 0,2,2
Blit 1,10,10
Blit 0,4,100
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ROTATE

```
Mode(s):   Amiga/Blitz
Statement: rotate a shape object
Syntax:    Rotate SHAPE#,ANGLE_RATIO
```

The ROTATE statement is used to rotate a shape object. The ANGLE_RATIO parameter specifies the size of clockwise rotation to be applied, in the range 0-1:

Table 8.3 : ROTATE angle ratios

```
ANGLE_RATIO  Size of rotation
=============================
0.00         0 degrees
0.25         90 degrees
0.50         180 degrees
0.75         270 degrees
1.00         360 degrees
```

Here is an example:

```
; *** Shape Rotation
; *** Filename - Rotate.bb2

PalRGB 0,1,15,15,15
BLITZ
BitMap 0,320,256,1
BitMapOutput 0
Locate 1,1
NPrint "Rotated, I'm sure"
Box 7,5,144,17,1
GetaShape 0,7,5,144,17
Cls
For A=1 To 4
  CopyShape 0,A
  Rotate A,A/10
  Blit A,50+(A*20),50
Next A
Slice 0,44,1
```

```
Use Palette 0
Show 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.2.7 Cookiecuts

AUTOCOOKIE

```
Mode(s):   Amiga/Blitz
Statement: toggle cookiecut mode
Syntax:    AutoCookie On/Off
```

When shape objects are used by any of the blitting commands they usually require the presence of a "cookiecut". These cookiecuts do not affect the appearance or qualities of a shape object, but they do consume Chip memory. When a shape is created using the LOADSHAPE or GETASHAPE commands, a cookiecut is automatically created for it. This may be turned off using the AUTOCOOKIE OFF statement and is primarily of use for shapes used as gadgets or in menus. Example:

```
; *** AutoCookie example
; *** Filename - AutoCookie.bb2

Screen 0,3
ScreensBitMap 0,0
For A=7 To 1 Step -1
  Circlef 16,32,A*2,A
Next
GetaShape 0,0,16,32,32
SaveShape 0,"RAM:SHAPE"
Cls
AutoCookie Off
LoadShape 0,"RAM:SHAPE"
ShapeGadget 0,148,50,0,1,0
TextGadget 0,140,180,0,2," Quit "
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=2
; *** Return to Blitz Basic 2 editor
End
```

MAKECOOKIE

```
Mode(s):   Amiga/Blitz
Statement: create a cookiecut for a shape
Syntax:    MakeCookie SHAPE#
```

MAKECOOKIE is used to create a cookiecut for a shape object. Here's an example:

```
; *** MakeCookie example
; *** Filename - MakeCookie.bb2

Screen 0,3
ScreensBitMap 0,0
For A=7 To 1 Step -1
  Circlef 16,32,A*2,A
Next
GetaShape 0,0,16,32,32
SaveShape 0,"RAM:SHAPE"
Cls
AutoCookie Off
LoadShape 0,"RAM:SHAPE"
; *** Try removing the next line
MakeCookie 0
For B=1 To 100
  Blit 0,Rnd(260)+30,Rnd(150)+30
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.3 Blitting

The following section covers all of the commands that are used to draw shapes on to BitMaps using the Amiga's Blitter chip.

As has been explained, there are three main blitting techniques: BLIT, QBLIT and BBLIT.

BLIT is the simplest blitting technique. It is primarily of use when displaying static images as it has no provision for the movement of shapes.

QBLIT does allow for movement, however it does corrupt any background graphics present. It is useful for animating shapes on blank backgrounds or on dual-playfield displays (where the shapes are displayed on one playfield and the background is held on another).

The great thing about BBLIT is that you don't have to worry about shapes corrupting your background graphics. Because the background is held in a special storage buffer, Blitz automatically redraws the background BitMap every time the shapes are moved.

8.3.1 A simple blit

BLIT

```
Mode(s):   Amiga/Blitz
Statement: draw a shape object on a BitMap
Syntax:    Blit SHAPE#,X,Y[,EXCESS]
```

The BLIT statement is the simplest of the blitting commands. Is is used to draw a shape object (SHAPE#) on the currently used BitMap at the co-ordinates specified by the X and Y parameters.

If the optional EXCESS parameter is included then any excess bitplanes may be set on or off. This is primarily of use when a shape object has fewer bitplanes than the BitMap on which it is displayed. EXCESS allows you to specify an on/off value for the excess bitplanes (i.e. the bitplanes beyond those altered by the shape):

Table 8.4 : The EXCESS parameter

```
Bit  On/Off value for...
========================
0    First bitpane
1    Second bitplane
2    Third bitplane
3    Fourth bitplane
4    Fifth bitplane
```

The BLITMODE statement may be used to alter the output of the BLIT statement. Here is a full example:

```
; *** Blit example
; *** Filename - Blit.bb2

BLITZ
BitMap 0,640,256,3
Slice 0,44,320,256,$fff8,3,8,8,640,640
Show 0
For A=1 To 15
  ColSplit 1,A,A,A,99+A
Next A
RGB 0,0,0,0
RGB 1,15,15,15
Boxf 20,20,40,40,1
GetaShape 0,20,20,40,40
Cls 0
For X=0 To 320
  VWait
  Blit 0,X,100
Next X
; *** Wait for a mouse click
```

```
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

CLIPBLIT

```
Mode(s):   Amiga/Blitz
Statement: draw a shape object on a BitMap
Syntax:    ClipBlit SHAPE#,X,Y
```

CLIPBLIT works identically to the BLIT statement, except it will clip the shape object to the inside of the currently used BitMap. For example:

```
; *** ClipBlit example
; *** Filename - ClipBlit.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
RGB 0,0,0,0
RGB 1,15,0,15
Boxf 20,20,40,40,1
GetaShape 0,20,20,40,40
Cls 0
; *** If you replace this line with
; *** a normal blit command then
; *** Blitz will generate a "Coords
; *** outside of BitMap" error
ClipBlit 0,310,100
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

BLOCK

```
Mode(s):   Amiga/Blitz
Statement: draw a shape object on a BitMap
Syntax:    Block SHAPE#,X,Y
```

This is an extremely fast version of the BLIT statement. However, BLOCK should only be used with shapes that are 16,32,48,64 etc. pixels wide and that are blitted to an x co-ordinate of 0,16,32,48,64 etc. (i.e. divisible by 16). The height and y co-ordinate of the shape are not limited by the BLOCK statement. Here is an example:

```
; *** Block example
; *** Filename - Block.bb2

Dim MAP(10,5)
BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
For A=0 To 7
  Boxf 9+A,9+A,26-A,26-A,A
Next A
GetaShape 0,10,10,26,26
Cls
Restore DAT
For Y=1 To 5
  For X=1 To 10
    Read MAP(X,Y)
    If MAP(X,Y)=1
      Block 0,X*16,Y*16
    End If
  Next X
Next Y
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End

DAT:
Data 1,0,0,1,0,0,1,1,1,0
Data 1,0,0,1,0,0,0,1,0,0
Data 1,1,1,1,0,0,0,1,0,0
Data 1,0,0,1,0,0,0,1,0,0
Data 1,0,0,1,0,0,1,1,1,0
```

8.3.2 Blit modes

BLITMODE

```
Mode(s):   Amiga/Blitz
Statement: change Blit mode
Syntax:    BlitMode BLTCON0
```

The BLITMODE statement is used to alter the output of the BLIT statement when drawing shape objects onto BitMaps:

Table 8.5 : Blit modes

```
BLTCON0     Description
=====================================================
CookieMode  Shape drawn normally (default mode)
EraseMode   Shape erases destination BitMap
InvMode     Shape inversed on destination BitMap
SolidMode   Shape drawn as a solid area of one colour
```

COOKIEMODE

```
Mode(s):  Amiga/Blitz
Function: change Blit mode to default
Syntax:   BlitMode CookieMode
```

The COOKIEMODE function returns a value which may be used by one of the commands involved in blitting modes. Using COOKIEMODE as a blitting mode will cause a shape to be blitted normally onto a BitMap.

ERASEMODE

```
Mode(s):  Amiga/Blitz
Function: change Blit mode to erase mode
Syntax:   BlitMode EraseMode
```

The ERASEMODE function returns a value which may be used by one of the commands involved in blitting modes. Using ERASEMODE as a blitting mode will cause a shape to erase a section of a BitMap corresponding to the outline of the shape.

INVMODE

```
Mode(s):  Amiga/Blitz
Function: change Blit mode to inverse mode
Syntax:   BlitMode InvMode
```

The INVMODE function returns a value which may be used by one of the commands involved in blitting modes. Using INVMODE as a blitting mode will cause a shape to invert a section of a BitMap corresponding to the outline of the shape.

SOLIDMODE

```
Mode(s):  Amiga/Blitz
Function: change Blit mode to one-colour mode
Syntax:   BlitMode SolidMode
```

The SOLIDMODE function returns a value which may be used by one of the commands involved in blitting modes. Using SOLIDMODE as a blitting mode will cause a shape to overwrite a section of a BitMap corresponding to the outline of the shape.

Here is a full example which demonstrates the various blitting modes:

```
; *** BlitMode example
; *** Filename - BlitMode.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
RGB 0,0,0,0
RGB 1,15,0,15
Circlef 32,32,28,5
Box 15,10,49,54,6
GetaShape 0,0,0,64,64
Boxf 0,0,250,130,4
BitMapOutput 0
; *** Default mode
BlitMode CookieMode
Blit 0,0,0
; *** Erase mode
BlitMode EraseMode
Blit 0,160,0
; *** Inverse mode
BlitMode InvMode
Blit 0,0,100
; *** Solid mode
BlitMode SolidMode
Blit 0,160,100
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.3.3 Queue blits

The correct procedure for creating a Queue blit is as follows:

1. Define a queue using QUEUE
2. Blit the shapes using QBLIT
3. Erase the shapes using UNQUEUE

QUEUE

```
Mode(s):   Amiga/Blitz
Statement: create a queue object
Syntax:    Queue QUEUE#,ITEMS
```

QBLIT

```
Mode(s):   Amiga/Blitz
Statement: draw a shape object on a BitMap
Syntax:    QBlit QUEUE#,SHAPE#,X,Y[,EXCESS]
```

UNQUEUE

```
Mode(s):   Amiga/Blitz
Statement: erase all previously QBlitted items
Syntax:    Unqueue QUEUE#[,BITMAP#]
```

The QUEUE statement defines a queue object for use with the QBLIT and UNQUEUE statements The ITEMS parameter specifies the maximum number of shapes the queue is capable of remembering.

QBLIT is used to draw a shape onto the currently used BitMap. It also stores the size and co-ordinates of the shape (consult the BLIT statement for parameter information).

The UNQUEUE statement is used to erase all remembered shapes in a queue. If the optional BITMAP# parameter is included then items may be erased by way of replacement from another BitMap.

Here's an example:

```
; *** QBlit example
; *** Filename - QBlit.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Boxf 10,10,20,20,6
```

```
GetaShape 0,10,10,20,20
Cls 0
; *** Define a queue for 10 shapes
Queue 0,10
For X=32 To 300
  VWait
  ; *** Erase shapes
  UnQueue 0
  ; *** Draw shapes
  For Y=1 To 10
    QBlit 0,0,X,10+Y*16
  Next Y
Next X
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FLUSHQUEUE

```
Mode(s):   Amiga/Blitz
Statement: erase a queue
Syntax:    FlushQueue QUEUE#
```

FLUSHQUEUE is used to erase a queue object, causing the next UNQUEUE statement to have no effect.

QBLITMODE

```
Mode(s):   Amiga/Blitz
Statement: change QBlit mode
Syntax:    QBlitMode BLTCON0
```

The QBLITMODE statement is used to alter the output of the QBLIT statement, when drawing shape objects onto BitMaps. It works identically to the BLITMODE statement:

Table 8.6 : QBlit modes

```
BLTCON0     Description
=====================================================
CookieMode  Shape drawn normally (default mode)
EraseMode   Shape erases destination BitMap
InvMode     Shape inversed on destination BitMap
SolidMode   Shape drawn as a solid area of one colour
```

8.3.4 Buffer blits

The correct procedure for creating a Buffer blit is as follows:

1. Define a storage buffer using BUFFER

2. Blit the shapes using BBLIT

3. Erase the shapes using UNBUFFER

BUFFER

```
Mode(s):   Amiga/Blitz
Statement: create a buffer object
Syntax:    Buffer BUFFER#,LENGTH
```

BBLIT

```
Mode(s):   Amiga/Blitz
Statement: draw a shape object on a BitMap
Syntax:    BBlit BUFFER#,SHAPE#,X,Y[,EXCESS]
```

UNBUFFER

```
Mode(s):   Amiga/Blitz
Statement: erase all previously BBlitted items
Syntax:    UnBuffer BUFFER#
```

The BUFFER statement is used to create a buffer object. Buffers differ from queues in their ability to preserve background graphics.

The LENGTH parameter specifies the memory, in bytes, to be used as temporary storage for the preservation of background graphics. The value of this parameter varies depending upon the size and maximum number of shapes to blit. A LENGTH of 16384 bytes (the default) should be enough, but this may be increased if you get "Buffer Overflow" error messages.

The BBLIT statement is used to draw a shape onto the currently used BitMap, and preserve the overwritten area into a previously initialised buffer (consult the BLIT statement for parameter information).

UNBUFFER simply replaces areas on a BitMap overwritten by the BBLIT statement.

Here is a complete example:

```
; *** BBlit example
; *** Filename - BBlit.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Boxf 10,10,20,20,7
GetaShape 0,10,10,20,20
Cls 0
For A=1 To 100
  Circlef Rnd(320),Rnd(200),Rnd(30)+10,Rnd(5)+1
Next A
; *** Create storage buffer
Buffer 0,16384
For X=32 To 300
  VWait
  ; *** Restore background
  UnBuffer 0
  For Y=1 To 10
    ; *** Draw shapes
    BBlit 0,0,X,10+Y*16
  Next Y
Next X
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FLUSHBUFFER

```
Mode(s):   Amiga/Blitz
Statement: erase a buffer
Syntax:    FlushBuffer BUFFER#
```

FLUSHBUFFER erases a buffer, causing the next UNBUFFER statement to have no effect.

BBLITMODE

```
Mode(s):   Amiga/Blitz
Statement: change BBlit mode
Syntax:    BBlitMode BLTCON0
```

The BBLITMODE statement is used to alter the output of the BBLIT statement, when drawing shape objects onto BitMaps. It works identically to the BLITMODE statement:

Table 8.7 : BBlit modes

```
BLTCON0     Description
=====================================================
CookieMode  Shape drawn normally (default mode)
EraseMode   Shape erases destination BitMap
InvMode     Shape inversed on destination BitMap
SolidMode   Shape drawn as a solid area of one colour
```

For example:

```
; *** BBlitMode example
; *** Filename - BBlitMode.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Boxf 10,10,20,20,7
GetaShape 0,10,10,20,20
Cls 0
For A=1 To 100
  Circlef Rnd(320),Rnd(200),Rnd(30)+10,Rnd(5)+1
Next A
BBlitMode InvMode
Buffer 0,16384
For X=32 To 300
  VWait
  UnBuffer 0
  For Y=1 To 10
    BBlit 0,0,X,10+Y*16
  Next Y
Next X
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

8.3.5 Stencil blits

The blitting technique which we haven't covered so far is the stencil blit. Stencils allow you to move shapes between background and foreground graphics to produce an illusion of depth.

Here is the correct procedure:

1. Draw a BitMap comprised of both foreground and background graphics.
2. Create a stencil with only the foreground graphics on it, using either STENCIL or SBLIT.
3. BBLIT the shapes.
4. Display the foreground graphics on top of the shapes using SHOWSTENCIL.

STENCIL

```
Mode(s):   Amiga/Blitz
Statement: create a stencil object
Syntax:    Stencil STENCIL#,BITMAP#
```

SHOWSTENCIL

```
Mode(s):   Amiga/Blitz
Statement: show stencil on a BitMap
Syntax:    ShowStencil BUFFER#,STENCIL#
```

The STENCIL statement creates a stencil object containing the contents of a BitMap.

SHOWSTENCIL is used to display the stencil object on a BitMap. Here is an example:

```
; *** Stencil example
; *** Filename - Stencil.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Circlef 30,30,30,2
GetaShape 0,0,0,61,60
Cls
; *** Draw mountain background
Y=200 : LAND=3 : DI=-1
For X=0 To 320
  D=Int(Rnd(LAND))
  If D=1 Then DI=-1
  If D=2 Then DI=1
  Let Y+DI
  If Y<0 Then Y=0
  If Y>256-1 Then Y=255
  Line X,256,X,Y-10,4
Next X
; *** Store mountain in memory
Stencil 0,0
Buffer 0,16384
For X=20 To 250
  VWait
  UnBuffer 0
  ; *** Show shape
  BBlit 0,0,X,150
  ; *** Replace mountain on top
  ShowStencil 0,0
Next X
```

```
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SBLIT

```
Mode(s):   Amiga/Blitz
Statement: draw a shape object on a BitMap and update stencil
Syntax:    SBlit STENCIL#,SHAPE#,X,Y[,EXCESS]
```

SBLIT works identically to the BLIT statement and also updates the specified stencil. This is an easy way to render foreground graphics to a BitMap. Try the following example:

```
; *** SBlit example
; *** Filename - SBlit.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Boxf 0,0,20,20,6
GetaShape 0,0,0,20,20
Boxf 0,0,20,20,7
GetaShape 1,0,0,20,20
Cls
; *** Set stencil
Stencil 0,0
For A=1 To 100
  Circlef Rnd(320),Rnd(256),Rnd(20)+5,Rnd(4)+1
Next A
; *** Update stencil and add shapes
For B=1 To 50
  SBlit 0,1,Rnd(280)+20,Rnd(180)+20
Next B
Buffer 0,16384
For X=20 To 300
  VWait
  UnBuffer 0
  ; *** Show shapes
  BBlit 0,0,X,50
  BBlit 0,0,X,100
  BBlit 0,0,X,150
  ; *** Replace stencil
  ShowStencil 0,0
Next X
; *** Wait for a mouse click
MouseWait
```

```
  ; *** Return to Blitz Basic 2 editor
  End
```

SBLITMODE

```
Mode(s):   Amiga/Blitz
Statement: change SBlit mode
Syntax:    SBlitMode BLTCON0
```

The SBLITMODE statement is used to alter the output of the SBLIT statement, when drawing shape objects onto BitMaps. It works identically to the BLITMODE statement:

Table 8.8 : SBlit modes

```
BLTCON0     Description
=====================================================
CookieMode  Shape drawn normally (default mode)
EraseMode   Shape erases destination BitMap
InvMode     Shape inversed on destination BitMap
SolidMode   Shape drawn as a solid area of one colour
```

8.4 Detecting Collisions

Virtually every computer game ever created uses collision detection to some extent. If the aliens didn't explode, or the cars didn't crash then there would be little point in playing. The secret of good collision detection lies with its accuracy: if the collision detection is too accurate then the player will die unneccessarily, and if the detection is too lenient then the player will always win. Striking the right balance between the two is easy with Blitz Basic and its comprehensive command set.

This entire section is devoted to the finer points of collision detection.

8.4.1 Colours and sprites

SETCOLL

```
Mode(s):   Amiga/Blitz
Statement: set collision detection to between a screen colour and sprite
Syntax:    c=SetColl COLOUR,BITPLANES[,PLAYFIELD]
```

This statement sets sprite/BitMap collisions to between sprites and individual screen colours. SETCOLL allows you to specify a single colour (the COLOUR parameter) which, when present in a BitMap, and in contact with a sprite, will cause a collision. The BITPLANES parameter specifies the number of bitplanes in the currently used BitMap (since SETCOLL involves sprites, this figure should be either 2 or 4). SETCOLL does not detect the actual collisions; it is used in conjunction with the PCOLL statement to control the finer points of collision detection.

The optional PLAYFIELD parameter should be included when dual-playfield Slices are being used. If PLAYFIELD is set to (1), then COLOUR refers to a colour on the foreground BitMap, and a PLAYFIELD value of (0) refers to a colour on the background BitMap. Try the following example, in which a collision occurs when the sprite collides with colour 15:

```
; *** SetColl example
; *** Filename - SetColl.bb2

BitMap 0,320,256,4
BitMapOutput 0
Boxf 0,0,7,7,3
GetaShape 0,0,0,8,8
GetaSprite 0,0
Free Shape 0
Cls
BLITZ
Slice 0,44,4
Show 0
For A=1 To 100
  Plot Rnd(320),Rnd(256),Rnd(14)+1
Next A
Boxf 80,100,250,200,15
SetColl 15,4
X=50: Y=50
X1=4 : Y1=4
Repeat
  VWait
  DoColl
  Locate 0,0
  If PColl(0)
    Print "Collision"
  Else
    Print "         "
  EndIf
  ShowSprite 0,X,Y,1
  Let X+X1 : Let Y+Y1
  If X<=10 OR X>=300 Then X1=-X1
  If Y<=20 OR Y>=200 Then Y1=-Y1
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

SETCOLLODD

```
Mode(s):   Amiga/Blitz
Statement: set collision detection between odd screen colours and sprites
Syntax:    SetCollOdd BITPLANES
```

The SETCOLLODD statement sets sprite/BitMap collisions to between sprites and the odd colours of the colour palette (eg. 1,3,5,7,9 etc.). The BITPLANES parameter should be set to the number of bitplanes in the current BitMap (either 2 or 4). Here is an example:

```
; *** SetCollOdd example
; *** Filename - SetCollOdd.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
Boxf 0,0,20,20,1
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
; *** Set odd colours to white
For COLS=1 To 16 Step 2
  RGB COLS,15,15,15
Next COLS
For A=1 To 30
  X=Rnd(320)
  Y=Rnd(200)
  Boxf X,Y,X+20,Y+20,Rnd(15)
Next A
; *** Set collision type
SetCollOdd
Mouse On
Pointer 0,0
Repeat
  VWait
  DoColl
  Locate 0,0
  ; *** Detect if white square touched
  If PColl(0)
    Print "Collision"
  Else
    Print "         "
  EndIf
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

SETCOLLHI

```
Mode(s):   Amiga/Blitz
Statement: set collision detection to between upper palette and sprites
Syntax:    SetCollHi BITPLANES
```

This statement sets sprite/BitMap collisions to between sprites and the upper-half of the colour palette:

Table 8.9 : The upper-half of a colour palette

```
Bitplanes  Colours  Upper-half
==============================
2          4        3+
4          16       8+
```

The BITPLANES parameter should be set to the number of bitplanes in the current BitMap. For example:

```
; *** SetCollHi example
; *** Filename - SetCollHi.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
Boxf 0,0,20,20,1
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
For A=1 To 8
  Circlef Rnd(320),Rnd(200),Rnd(20)+10,Rnd(9)+7
Next A
SetCollHi 4
Mouse On
Pointer 0,0
Repeat
  VWait
  DoColl
  Locate 0,0
  If PColl(0)
    Print "Collision"
  Else
    Print "         "
  EndIf
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

8.4.2 Executing collision detection

DOCOLL

```
Mode(s):   Blitz
Statement: execute collision detection
Syntax:    DoColl
```

The DOCOLL statement executes sprite/BitMap collision detection and must be called before each PCOLL and SCOLL statement. Before DOCOLL can be used in conjunction with PCOLL, the type of BitMap collisions to be detected must have been specified using either SETCOLL, SETCOLLODD or SETCOLLHI.

Table 8.10 : The detection methods which need DoColl

Detection method	DoColl?
PCOLL	Y
SCOLL	Y
SHAPESHIT	N
BLITCOLL	N
SHAPESPRITEHIT	N
SPRITESHIT	N
RECTSHIT	N

Try the following example:

```
; *** DoColl example
; *** Filename - DoColl.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
Boxf 0,0,20,20,13
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
For A=1 To 20
  Circlef Rnd(320),Rnd(200),Rnd(20)+10,Rnd(7)+9
Next A
SetCollHi 4
For X=10 To 300
  VWait
  ShowSprite 0,X,100,0
  DoColl
```

```
    Locate 0,0
    If PColl(0)
      Print "Collision"
    Else
      Print "         "
    EndIf
  Next X
  ; *** Return to Blitz Basic 2 editor
  End
```

8.4.3 Collision checking

PCOLL

```
Mode(s):  Blitz
Function: check for collision between a particular sprite and screen colour
Syntax:   p=PColl(SPRITE#)
```

PCOLL checks for collisions between a sprite and any BitMap graphics. If a collision has occured then (-1) is returned, otherwise (0) is returned. PCOLL must follow a DCOLL statement. See previous example.

8.4.4 Sprite colisions

SCOLL

```
Mode(s):  Blitz
Function: check for collision between 2 sprites
Syntax:   s=SColl(SPRITE1#,SPRITE2#)
```

The SCOLL function returns the collision status between two sprites (SPRITE1# and SPRITE2#). If the sprites have collided then (-1) is returned, otherwise (0) is returned. SCOLL must follow a DCOLL statement. Example:

```
; *** SColl example
; *** Filename - SColl.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
Boxf 0,0,20,20,13
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
For A=1 To 100
```

```
  Plot Rnd(320),Rnd(200),Rnd(7)+9
Next A
X2=300
For X=10 To 300
  VWait
  DoColl
  ShowSprite 0,X,100,0
  ShowSprite 0,X2,100,4
  Locate 0,0
  If SColl(0,4)
    Print "BOOM!!!"
  Else
    Print "       "
  EndIf
  Let X2-1
Next X
; *** Return to Blitz Basic 2 editor
End
```

8.4.5 Shape collisions

SHAPESHIT

```
Mode(s):  Amiga/Blitz
Function: check for collision between 2 shapes
Syntax:   ShapesHit(SHAPE1#,X1,Y1,SHAPE2#,X2,Y2)
```

This function returns the collision status of two rectangular shape areas. The X and Y parameters are the coordinates of the shapes to check. If the two shapes overlap then (-1) wil be returned, otherwise (0) will be returned. For example:

```
; *** ShapesHit example
; *** Filename - ShapesHit.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
Boxf 0,0,20,20,1
Boxf 5,5,15,15,2
GetaShape 0,0,0,20,20
Cls
For A=1 To 100
  Plot Rnd(320),Rnd(200),Rnd(7)+9
Next A
X2=300
Buffer 0,16384
```

```
For X=10 To 300
  VWait
  UnBuffer 0
  BBlit 0,0,X,100,0
  BBlit 0,0,X2,100,4
  Locate 0,0
  If ShapesHit(0,X,100,0,X2,100)
    Print "BOOM!!!"
  Else
    Print "       "
  EndIf
  Let X2-1
Next X
; *** Return to Blitz Basic 2 editor
End
```

BLITCOLL

```
Mode(s):  Amiga/Blitz
Function: return the collision status of a shape
Syntax:   b=BlitColl(SHAPE#,X,Y)
```

The BLITCOLL function provides a fast way of testing the collision status of a shape (SHAPE#). A collision occurs if the shape object touches any pixel on the currently used BitMap that is not of colour zero. If a collision occurs then (-1) is returned, otherwise (0) is returned. For example:

```
; *** BlitColl example
; *** Filename - BlitColl.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
For COLS=1 To 8
  Boxf 0+COLS,0+COLS,20-COLS,20-COLS,COLS
Next COLS
GetaShape 0,0,0,20,20
Cls
For A=1 To 15
  Circlef Rnd(320),Rnd(200),Rnd(20)+10,Rnd(7)+9
Next A
Buffer 0,16384
For X=10 To 300
  VWait
  UnBuffer 0
  Locate 0,0
  If BlitColl(0,X,100)
```

```
    Print "Collision detected"
  Else
    Print "                  "
  EndIf
  BBlit 0,0,X,100,0
Next X
; *** Return to Blitz Basic 2 editor
End
```

8.4.6 Shape and sprite collisions

SHAPESPRITEHIT

```
Mode(s):  Amiga/Blitz
Function: check for collision between a shape and a sprite
Syntax:   s=ShapeSpriteHit(SHAPE#,X1,Y1,SPRITE#,X2,Y2)
```

The SHAPESPRITEHIT returns the collision status of a rectangular sprite area and a rectangular shape area. The X and Y parameters are the coordinates of the sprite/shape to check. If the sprite and the shape overlap then (-1) wil be returned, otherwise (0) will be returned:

```
; *** ShapeSpriteHit example
; *** Filename - ShapeSpriteHit.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
For COLS=1 To 8
  Boxf 0+COLS,0+COLS,20-COLS,20-COLS,COLS
Next COLS
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
Buffer 0,16384
X2=300
For X=10 To 300
  VWait
  UnBuffer 0
  Locate 0,0
  If ShapeSpriteHit(0,X,100,0,X2,100)
    Print "Collision detected"
  Else
    Print "                  "
  EndIf
  BBlit 0,0,X,100,0
  ShowSprite 0,X2,100,0
  Let X2-1
Next X
; *** Return to Blitz Basic 2 editor
End
```

8.4.7 Sprite area collisions

SPRITESHIT

```
Mode(s):  Amiga/Blitz
Function: check for collision between 2 rectangular sprite areas
Syntax:   s=SpritesHit(SPRITE1#,X1,Y1,SPRITE2#,X2,Y2)
```

This function returns the collision status of two rectangular sprite areas. The X and Y parameters are the coordinates of the sprites to check. If the two sprites overlap then (-1) wil be returned, otherwise (0) will be returned. For example:

```
; *** SpritesHit example
; *** Filename - SpritesHit.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
Boxf 0,0,20,20,13
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
For A=1 To 100
  Plot Rnd(320),Rnd(200),Rnd(7)+9
Next A
X2=300
For X=10 To 300
  VWait
  ShowSprite 0,X,100,0
  ShowSprite 0,X2,100,4
  Locate 0,0
  If SpritesHit(0,X,100,0,X2,100)
    Print "BOOM!!!"
  Else
    Print "       "
  EndIf
Let X2-1
Next X
; *** Return to Blitz Basic 2 editor
End
```

8.4.8 Rectangular area collisions

RECTSHIT

```
Mode(s):  Amiga/Blitz
Function: check for collision between 2 rectangular areas
Syntax:   r=RectsHit(X1,Y1,WIDTH1,HEIGHT1,X2,Y2,WIDTH2,HEIGHT2)
```

The RECTSHIT function returns the collision status of two rectangular areas. X1,Y1,WIDTH1 and HEIGHT1 are the coordinates of the first rectangular area and X2,Y2,WIDTH2 and HEIGHT2 are the coordinates of the second rectangular area. If the two areas overlap (or collide) then (-1) will be returned, otherwise (0) will be returned. Try the following example:

```
; *** RectsHit example
; *** Filename - RectsHit.bb2

BLITZ
BitMap 0,320,256,4
BitMapOutput 0
Slice 0,44,4
Show 0
GetaShape 0,0,0,20,20
GetaSprite 0,0
Cls
X2=300
Y2=100
For X=10 To 300
  VWait
  Locate 0,0
  If RectsHit(0,100,30+X,120,X2,Y2,X2+50,Y2+50)
    Print "Collision detected"
    MouseWait
    End
  EndIf
  Boxf 0+X,100,30+X,150,8
  Boxf X2,Y2,X2+50,Y2+50,3
  Let X2-1
Next X
; *** Return to Blitz Basic 2 editor
End
```

8.5 End-of-Chapter summary

A sprite is a graphical object which is moved by the Amiga's hardware and does not corrupt background graphics. Sprites are initialised by either loading them from disk, or by converting a shape object into a sprite object using the GETASPRITE statement.

Shapes are objects drawn, or blitted, by the Amiga's Blitter chip. There are three main blitting techniques: BLIT, QBLIT and BBLIT.

Collisions between colours and sprites are defined using the SETCOLL statement.

The DOCOLL statement executes sprite/BitMap collision detection and must be called before each PCOLL and SCOLL statement.

The SCOLL function returns the collision status between two sprites.

PCOLL checks for collisions between a sprite and any BitMap graphics.

Collisions involving shapes are tested using the SHAPESHIT and BLITCOLL functions.

Collisions between sprites are tested using SPRITESHIT.

Collisions between shapes and sprites are tested using the SHAPESPRITEHIT function.

Collisions between two rectangular areas are tested using RECTSHIT.

Chapter 9 : Audio

9.1 Pump up the volume

VOLUME

```
Mode(s):   Amiga/Blitz
Statement: control sound volume
Syntax:    Volume MASK,VOL1[,VOL2][,VOL3][,VOL4]
```

The VOLUME statement controls the volume of sound which passes through one or more of the Amiga's four sound channels. The MASK parameter specifies which of the Amiga's four audio channels the sound should be played through, from one to 15:

Table 9.1 : Audio masks

MASK	Channel 0	Channel 1	Channel 2	Channel 3
1	on	off	off	off
2	off	on	off	off
3	on	on	off	off
4	off	off	on	off
5	on	off	on	off
6	off	on	on	off
7	on	on	on	off
8	off	off	off	on
9	on	off	off	on
10	off	on	off	on
11	on	on	off	on
12	off	off	on	on
13	on	off	on	on
14	off	on	on	on
15	on	on	on	on

The optional VOL parameters are used to set the volume of the four audio channels. Volume settings should be in the range zero (silence) to 64 (loudest). The first VOL parameter specifies the volume of the first "on" audio channel, the sencond VOL for the next "on" audio channel and so on.

If any VOL parameters are not included then their associated channel will be given a volume of 64.

For example:

```
; *** Pump up the ...
; *** Filename - Volume.bb2

; *** Initialise sound object
```

```
InitSound 0,32
; *** Create sound data using sine waveform
For A=0 To 31
  SoundData 0,A,Sin(A*150)*127
Next A
; *** Play sound
LoopSound 0,1
; *** Fade out volume
For V=64 To 0 Step -1
  VWait
  Volume 1,V
Next V
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FILTER

```
Mode(s):   Amiga/Blitz
Statement: toggle the Amiga's low pass audio filter
Syntax:    Filter On/Off
```

The FILTER statement is used to toggle sound distortion with the Amiga's audio filter. Some people prefer FILTER to be set to ON, whilst other musical connaisseurs prefer it to remain OFF - it really is a matter of preference. Here is an example:

```
; *** Engine sound
; *** Filename - Filter.bb2

; *** Initialise sound object
InitSound 0,32
; *** Create sound data using sine waveform
For A=0 To 31
  SoundData 0,A,Sin(A*50)*127
Next A
; *** Play sound 100 times
For B=0 To 100
  ; *** Toggle filter on
  Filter On
  ; *** Play sound
  Sound 0,15
  VWait 3
  ; *** Toggle filter off
  Filter Off
  ; *** Play sound
  Sound 0,15
Next B
```

```
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.2 Rave the waves

INITSOUND

```
Mode(s):   Amiga/Blitz
Statement: intialize a sound object
Syntax:    InitSound SOUND#,LENGTH[,PERIOD[,REPEAT]]
```

This statement is used to initialise a sound object. Sound objects can be simple sine or square waveforms, created with SOUNDDATA.

The LENGTH parameter specifies the length, in bytes, of the sound object. This must be less than 128K and an even number!

The optional PERIOD parameter, if included, allows you to specify the default pitch for the sound object.

The optional REPEAT parameter is used in conjunction with the LOOPSOUND statement. It specifies a position in the sound at which repeating should begin. Consult LOOPSOUND for more information.

Here's an example:

```
; *** InitSound example
; *** Filename - InitSound.bb2

; *** Initialise sound object
InitSound 0,32
; *** Create sound data using sine waveform
For A=0 To 31
  SoundData 0,A,Sin(A*2)*127
Next A
; *** Play sound (loop)
LoopSound 0,15
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SOUNDDATA

```
Mode(s):   Amiga/Blitz
Statement: define a wave form
Syntax:    SoundData SOUND#,OFFSET,DATA
```

SOUNDDATA is used to define the waveform of a sound object. It alters one byte of sound data at the specified OFFSET. The DATA parameter specifies the actual byte to be placed into the sound, and should be in the range -128 to +127. For example:

```
; *** SoundData example
; *** Filename - SoundData.bb2

; *** Initialise sound object
InitSound 0,32
; *** Create sound data using a waveform
For A=0 To 31
  If A<16
    SoundData 0,A,127
  Else
    SoundData 0,A,-128
  EndIf
Next
; *** Play sound (loop)
LoopSound 0,15
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.3 Samples

9.3.1 Playing samples from memory

LOADSOUND

```
Mode(s):   Amiga
Statement: load a sample into memory
Syntax:    LoadSound SOUND#,"FILENAME"
```

This statement loads a sample into memory. The sample should be in 8SVX IFF format, otherwise an error will be generated. Try this example:

```
; *** LoadSound example
; *** Filename - LoadSound.bb2

; *** Load sound sample from disk
LoadSound 0,"FILENAME.IFF"
; *** Play sound
Sound 0,1
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SOUND

```
Mode(s):   Amiga/Blitz
Statement: play a sample from memory
Syntax:    Sound SOUND#,MASK[,VOL1][,VOL2][,VOL3][,VOL4]
```

The SOUND statement is used to play a sample from memory. The MASK parameter specifies which of the Amiga's four audio channels the sample should be played through, from one to 15:

Table 9.2 : Audio masks

MASK	Channel 0	Channel 1	Channel 2	Channel 3
1	on	off	off	off
2	off	on	off	off
3	on	on	off	off
4	off	off	on	off
5	on	off	on	off
6	off	on	on	off
7	on	on	on	off
8	off	off	off	on
9	on	off	off	on
10	off	on	off	on
11	on	on	off	on
12	off	off	on	on
13	on	off	on	on
14	off	on	on	on
15	on	on	on	on

The optional VOL parameters are used to set the volume of the four audio channels. Volume settings should be in the range zero (silence) to 64 (loudest). The first VOL parameter specifies the volume of the first "on" audio channel, the sencond VOL for the next "on" audio channel and so on.

If any VOL parameters are not included then their associated channel will be given a volume of 64.

For example, the following syntax is used:

```
Sound 0,12,32,64
```

This would cause channels zero and one to be "off", and channels two and three to be "on". Because channels two and three are the first "on" channels, channel two would be given a volume setting of 32, and channel three a setting of 64.

If this syntax was used instead then channel three would be set to 32, as it is the only "on" audio channel:

```
Sound 0,8,32
```

Here is a full example:

```
; *** A Sound example
; *** Filename - Sound.bb2

; *** Load sound sample from disk
LoadSound 0,"FILENAME.IFF"
; *** Play sound
; *** (Channels 1 and 2 are half volume)
; *** (Channels 3 and 4 are full volume)
Sound 0,15,32,32,64,64
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

LOOPSOUND

```
Mode(s):   Amiga/Blitz
Statement: play and loop a sample from memory
Syntax:    LoopSound SOUND#,MASK[,VOL1][,VOL2][,VOL3][,VOL4]
```

This statement works identically to SOUND, except the sample will loop. Consult the LOADSOUND statement for parameter information. For example:

```
; *** LoopSound
; *** Filename - LoopSound.bb2

; *** Load sound sample from disk
LoadSound 0,"FILENAME.IFF"
```

```
; *** Play sound (loop)
LoopSound 0,15
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.3.2 Playing samples from disk

DISKPLAY

```
Mode(s):   Amiga
Statement: play a sample straight from disk
Syntax:    DiskPlay "FILENAME",MASK[,VOL1][,VOL2][,VOL3][,VOL4]
```

The DISKPLAY statement is used to play an 8SVX IFF sound sample straight from disk. It suspends program execution until the sample has stopped playing. DISKPLAY allows samples of any length, whereas LOADSOUND restricts its samples to 128K in length. Consult the LOADSOUND statement for parameter information. Example:

```
; *** DiskPlay example
; *** Filename - DiskPlay.bb2

; *** Load sound sample from disk and play
DiskPlay "FILENAME.IFF",1,64
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.3.3 Manipulating samples

DISKBUFFER

```
Mode(s):   Amiga/Blitz
Statement: set size of DiskPlay memory buffer
Syntax:    DiskBuffer LENGTH
```

This is used to set the size of the DISKPLAY memory buffer. The buffer is initially set to 1024 bytes, although this may be increased or decreased as needed. However, decreasing the memory buffer may cause a loss in sound quality. Here's an example:

```
; *** DiskBuffer example
; *** Filename - DiskBuffer.bb2

F$="A SAMPLE"
; *** Play sample normally
DiskPlay F$,15
; *** Wait for a mouse click
MouseWait
; *** Reduce buffer
DiskBuffer 128
; *** Play sample again (altered)
DiskPlay F$,15
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

PEEKSOUND

```
Mode(s):  Amiga/Blitz
Function: return the byte of a sample
Syntax:   PeekSound SOUND#,OFFSET
```

PEEKSOUND returns the byte of a sample at the specified offset of a sound object. For example:

```
; *** PeekSound example ** Filename - PeekSound.bb2
; *** Initialise sound object
InitSound 0,32
For A=0 To 31
  ; *** Set first half to max byte
  If A<16
    SoundData 0,A,127
  Else
    ; *** Set second half to min byte
    SoundData 0,A,-128
  EndIf
Next A
; *** Output all sample bytes
For B=0 To 31
  NPrint PeekSound(0,B)
Next B
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.4 Playing Tracker modules

A Tracker is a sequencing program which allows you to enter musical motes and arrange them to create a song, or module. The standard Tracker program on the Amiga is the Public Domain package Protracker. Protracker is based on the ageing Soundtracker program, a commercial piece of software which was unwittingly released into the Public Domain.

The latest version of Protracker is available from 17 Bit Software, at the following address:

```
17 Bit Software
1st Floor Offices
2/8 Market Street
Wakefield
West Yorkshire
WF1 1DH
Tel: (01924) 366982
```

LOADMODULE

```
Mode(s):   Amiga
Statement: load a Tracker module
Syntax:    LoadModule MODULE#,"FILENAME"
```

PLAYMODULE

```
Mode(s):   Amiga/Blitz
Statement: play a Tracker module
Syntax:    PlayModule MODULE#
```

To load a Tracker module into memory, use this statement. The MODULE# parameter is a unique identification value. This allows you to store more than one module in memory at once.

The PLAYMODULE statement is used to start a Tracker module playing from memory. MODULE# is the number of the module to play. For example:

```
; *** Loading a tracker module
; *** Filename - LoadModule.bb2

; *** Open a basic screen
Screen 0,3,"Module Master..."
; *** Load Tracker module from disk
LoadModule 0,"FILENAME.MOD"
; *** Start the module playing
PlayModule 0
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

STOPMODULE

```
Mode(s):   Amiga/Blitz
Statement: stop all Tracker modules
Syntax:    StopModule
```

The STOPMODULE statement is used to stop ALL Tracker modules currently being played.

FREE MODULE

```
Mode(s):   Amiga/Blitz
Statement: erase a Tracker module from memory
Syntax:    Free Module MODULE#
```

If you want to disguard a Tracker module from memory then use this statement. The MODULE# parameter specifies a module to erase. For example:

```
; *** Stop!
; *** Filename - StopModule.bb2

; *** Open another screen
Screen 0,3,"Module Master..."
; *** Load Tracker module from disk
LoadModule 0,"FILENAME.MOD"
; *** Start module playing
PlayModule 0
; *** Wait for a mouse click
MouseWait
; *** Stop module from playing
StopModule
; *** Remove module from memory
Free Module 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.5 Med modules

Med, or more recently OctaMed, is a superior music Tracker. Med modules are created in much the same way as Tracker ones, by playing notes on the Amiga's keyboard. Each note can be a different sample, and patterns of up to 64 steps can be created and pasted together to form a musical masterpiece.

Other notable options include:

- Sample editor
- Synthesised sound editor
- MIDI support
- On-line help

The latest incarnation of Med, namely OctaMed Pro V5, allows you to enter eight tracks of audio instead of four. This is achieved by playing two samples out of each audio channel. However, Blitz Basic does not currently support eight channel modules, so you are advised to stick with four.

OctaMed Pro V5 requires Kickstart 2.04 or later, and is available through Seasoft Computing, priced £30.00, from the following address:

Seasoft Computing Unit 3 Martello Enterprise Centre Courtwick Lane Littlehampton West Sussex England BN17 7PA Tel: (01903) 850378

9.5.1 Playing Med modules

LOADMEDMODULE

```
Mode(s):   Amiga
Statement: load a Med module
Syntax:    LoadMedModule MODULE#,"FILENAME"
```

The LOADMEDMODULE statement loads any four channel OctaMed module. The following commands support upto and including version 3 of the Amiganuts Med standard.

STARTMEDMODULE

```
Mode(s):   Amiga/Blitz
Statement: initialise a Med module in memory
Syntax:    StartMedModule MODULE#
```

STARTMEDMODULE is responsible for initialising the module including linking after it is loaded from disk using the LOADMEMODULE statement. It can also be used to restart a module from the beginning.

PLAYMED

```
Mode(s):   Amiga/Blitz
Statement: play a Med module
Syntax:    PlayMed
```

The PLAYMED statement plays the current Med module. It must be called every 50th of a second, either on an interrupt (#5), or after a VWAIT statement.

STOPMED

```
Mode(s):   Amiga/Blitz
Statement: stop the current Med module
Syntax:    StopMed
```

STOPMED will cause any Med module to stop playing.

Here is a full example which demonstrates the correct procedure for loading and playing a Med module:

```
; *** Playing a Med module
; *** Filename - PlayMed.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
Until Joyb(0)>0
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.5.2 Manipulating Med modules

SETMEDVOLUME

```
Mode(s):   Amiga/Blitz
Statement: set the volume of a Med module
Syntax:    SetMedVolume VOLUME
```

The SETMEDVOLUME statement changes the volume of a Med module. All music channels are affected by this statement. For example:

```
; *** Music fading
; *** Filename - SetMedVolume.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
Until Joyb(0)>0
; *** Fade out module volume
For A=64 To 1 Step -1
  VWait
  PlayMed
  SetMedVolume A
Next A
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

GETMEDVOLUME

```
Mode(s):  Amiga/Blitz
Function: return the current volume setting of an audio channel
Syntax:   g=GetMedVolume(CHANNEL)
```

This function returns the current volume setting of the specified audio channel. GETMEDVOLUME may be used to create graphic equalisers that move in time with the music. Try the following example:

```
; *** GetMedVolume example
; *** Filename - GetMedVolume.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
  ; *** Output volume of channel 0
  If Int(Rnd(100))=0
    A=GetMedVolume(0)
    NPrint "Volume 0 = ",A
  EndIf
Until Joyb(0)>0
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SETMEDMASK

```
Mode(s):   Amiga/Blitz
Statement: mask a Med audio channel
Syntax:    SetMedMask CHANNEL
```

SETMEDMASK allows the user to mask out an audio channel. The CHANNEL parameter specifies the number of a channel to mask, or silence. Try the following example:

```
; *** Masking example
; *** Filename - SetMedMask.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
; *** Mask all channels except channel 3
SetMedMask 0
SetMedMask 1
SetMedMask 2
Print "Channel 3 playing only"
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
Until Joyb(0)>0
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

JUMPMED

```
Mode(s):   Amiga/Blitz
Statement: change pattern being played in current Med module
Syntax:    JumpMed PATTERN
```

The JUMPMED statement is used to change the pattern being played in the current Med module. This is useful if you want the music to change in your games at different points. Say, for example, that you wanted a short piece of music to play once the player completed the game. You would write the music so that a few patterns (the end-game piece) are never played by the main module. These could then be jumped to, when required, by the JUMPMED statement. Here is an example:

```
; *** JumpMed example
; *** Filename - JumpMed.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
  ; *** Jump between module patterns
  If Int(Rnd(100))=0
    A=Int(Rnd(10)+1)
    NPrint "Jumping to pattern ",A
    JumpMed A
  EndIf
Until Joyb(0)>0
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

GETMEDNOTE

```
Mode(s):  Amiga/Blitz
Function: return current note playing through a channel
Syntax:   n=GetMedNote(CHANNEL)
```

This function returns the current note playing through the specified audio channel. Here is an example:

```
; *** GetMedNote example
; *** Filename - GetMedNote.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
  ; *** Output current note (channel 0)
  A=GetMedNote(0)
  Print A
Until Joyb(0)>0
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

GETMEDINSTR

```
Mode(s):  Amiga/Blitz
Function: return current instrument playing through a channel
Syntax:   i=GetMedInstr(CHANNEL)
```

GETMEDINSTR returns the current instrument playing through the specified audio channel:

```
; *** GetMedInstr example
; *** Filename - GetMedInstr.bb2

; *** Load Med module from disk
LoadMedModule 0,"MED_MODULE"
; *** Initialise Med module
StartMedModule 0
```

```
Repeat
  ; *** Play module every 50Hz
  VWait
  PlayMed
  ; *** Output instrument in a given channel
  If Int(Rnd(100))=0
    A=Int(Rnd(3)+1)
    B=GetMedInstr(A)
    NPrint "Instrument (channel ",A,") = ",B
  EndIf
Until Joyb(0)>0
; *** Stop module from playing
StopMed
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.6 Speech

One of the fun utilities provided with the Amiga was the narrator device; this allowed pre-AGA Amigas to "talk". For reasons known only to themselves, Commodore chose to remove the "speech" facility from Workbench 3.

A recent update has added speech to Blitz Basic 2, so that owners of all Amigas (including those equipped with the AGA chipset) can access this fabulous facility through BASIC.

9.6.1 Walkie Talkie

SPEAK

```
Mode(s):   Amiga
Statement: speak a string
Syntax:    Speak TEXT$
```

The SPEAK statement is used to pass a string of phonemes to the Amiga's voice synthesizer. SPEAK automatically converts any string to phonetics, so you don't need to worry about getting your hands dirty with the translation. Here is an example:

```
; *** Speak demo
; *** Filename - Speak.bb2

Repeat
  NPrint ""
  ; *** Input a string to speak
  NPrint "Please input some stuff:>"
  A$=Edit$(70)
  ; *** Talk!!!
  Speak A$
Until A$=""
; *** Return to Blitz Basic 2 editor
End
```

SETVOICE

```
Mode(s):   Amiga
Statement: set style of speech
Syntax:    SetVoice RATE,PITCH,EXPRESSION,SEX,VOLUME,FREQUENCY
```

SETVOICE can be used to alter the style of speech by changing the rate, pitch, expression, sex, volume and frequency of the Amiga's voice synthesizer:

Table 9.3 : SETVOICE parameters

Parameter	Description	Range	Default
RATE	Words per minute	40-400	150
PITCH	Baseline pitch in Hz	65-320	110
EXPRESSION	0=robot 1=natural 2=manual	0-2	1
SEX	0=male 1=female	0-1	0
VOLUME	Volume	0-64	64
FREQUENCY	Samples per second	0-22,200	22,200

Here is an example:

```
; *** A fine voice
; *** Filename - SetVoice.bb2

; *** Toggle audio filter on
Filter On
Repeat
  NPrint ""
  ; *** Input a string to speak
  NPrint "Enter some words or numbers to spell:>"
  A$=Edit$(70)
  A=1
  For B=0 To Len(A$)
    ; *** Split up string into characters
    B$=Mid$(A$,A,1)
    Let A+1
    VWait 5
    ; *** Alter audio voice randomly
    RATE=40+Rnd(360)
    PITCH=65+Rnd(255)
    SEX=Rnd(1)
    SetVoice RATE,PITCH,1,SEX,64,22200
    ; *** Speak next letter of string
    Speak B$
  Next B
  ; *** Speak entire string
  Speak A$
Until A$=""
; *** Return to Blitz Basic 2 editor
End
```

9.6.2 It's a foreign language

TRANSLATE$

```
Mode(s):  Amiga
Function: return the phonetic equivalent of a string
Syntax:   phonetic$=Translate$(TEXT$)
```

TRANSLATE$ returns the phonetic equivalent of a string. For example:

```
; *** Translate$ example
; *** Filename - Translate$.bb2

NPrint ""
NPrint "Enter a sentence:"
A$=Edit$(70)
NPrint"Phonetic =",Translate$(A$)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

PHONETICSPEAK

```
Mode(s):   Amiga
Statement: speak a phonetic string
Syntax:    PhoneticSpeak TEXT$
```

This statement is identical to the SPEAK statement, except the string to speak must contain legal phonemes. TEXT$ should be created by the TRANSLATE$ function. Try the following example:

```
; *** PhoneticSpeak example
; *** Filename - PhoneticSpeak.bb2

NPrint ""
NPrint "Enter a sentence:"
A$=Edit$(70)
PhoneticSpeak Translate$(A$)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

9.6 End-of-Chapter summary

The VOLUME statement controls the volume of sound which passes through one or more of the Amiga's four sound channels.

The FILTER statement is used to toggle sound distortion with the Amiga's audio filter.

INITSOUND and SOUNDDATA are used to create sound data from scratch.

Sound samples are played using LOADSOUND, SOUND and LOOPSOUND. Samples can be played straight from disk using the DISKPLAY statement.

Blitz Basic can play Tracker modules and four channel Med, or OctaMed, modules.

The Amiga can be made to "talk" using the SPEAK statement, and the style of this speech can be altered using SETVOICE.

Chapter 10 : Screens

This chapter explains how Intuition screens are created and manipulated by Blitz Basic 2.

A screen is an area of the display that shares the same attributes, such as size, resolution and colours.

The Amiga can have several screens open at once. However, unlike Slices, there are no limits placed upon how multiple screens may be arranged. Multiple screens can be positioned vertically on top of each other and may overlap.

10.1 Defining a screen

SCREEN

```
Mode(s):   Amiga
Statement: open an Intuition screen
Syntax:    Screen SCREEN#,MODE[,TITLE$]
Syntax 2:  Screen SCREEN#,X,Y,W,H,MODE,VIEWMODE,T$,D,B[,BITMAP#]
```

10.1.1 Syntax 1

The SCREEN statement is used to open an Intuition screen. The first syntax uses three parameters: SCREEN# (the screen number), MODE and the optional TITLE$ parameter.

The MODE parameter specifies the number of bitplanes for the screen, ranging from (1) to (6). The value you specify determines the number of colours that can be displayed on the screen, as shown in the following table (AGA screen modes are not currently supported by Blitz Basic 2):

Table 10.1 : Number of colours per bitplane

```
Bitplanes  Colours
==================
1          2
2          4
3          8
4          16
5          32
6          64
```

As with Slices, high-resolution screens can be opened by adding eight to this figure. High-resolution screens may use a maximum of four bitplanes (16 colours). Adding 16 to the MODE parameter creates an interlaced screen.

Note that the height of the screen will be 256 pixels on a PAL Amiga, or 200 pixels on an NTSC Amiga.

Here are some examples:

```
; *** Screen example 1
; *** Filename - Screen1.bb2

; *** Untitled low-res screen (3 bitplanes)
Screen 0,3
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Screen example 2
; *** Filename - Screen2.bb2

; *** Titled hi-res screen (2 bitplanes)
Screen 0,10,"Blitz"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.1.2 Syntax 2

The second SCREEN syntax is more involved and requires more parameters to operate:

Table 10.2 : SCREEN parameters

Parameter	Function
X	Horizontal position of top-left of screen
Y	Vertical position of top-left of screen
W	Width of screen (at least 320)
H	Height of screen
MODE	Number of bitplanes (up to 6 - 64/4096 colours)
VIEWMODE	Screen ViewMode
T$	Screen title
D	Detail pen colour
B	Block pen colour
[BITMAP#]	Attach BitMap to a screen

It is worth mentioning that screen widths must be a multiple of 16 and they are always at least the full width of the viewable area (a minimum of 320 pixels). The height of the screen will be 256 pixels on a PAL Amiga, or 200 pixels on an NTSC Amiga.

All of the other parameters are self-explanatory, except perhaps for VIEWMODE. VIEWMODE is a special parameter which enables the Blitz Basic programmer to create Half-Brite, HAM, Interlaced, High-resolution and Super-hi-res screens.

Table 10.3 : Screen ViewModes

```
VIEWMODE  Description
===================================================
$0000     Low-res
$0004     Interlace
$0080     Half-brite
$0800     HAM
$8000     Hi-res
$8020     Super-hi-res (AGA only - max 2 bitplanes)
```

```
; *** Low-resolution Screen example
; *** Filename - LowResScreen.bb2

Screen 0,0,100,320,100,3,$0000,"Low-res",1,2
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Hi-resolution Screen example
; *** Filename - HiResScreen.bb2

Screen 0,0,0,640,200,4,$8000,"Hi-res",1,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

```
; *** Super-hi-res Screen example
; *** Filename - SuperHiresScreen.bb2

Screen 0,0,0,1280,256,2,$8020,"Super-hi-res",1,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.1.3 Interlaced screens

Interlaced screens have twice the number of vertical lines as low-resolution screens. Like low-resolution mode, interlace format allows up to 64 colours to be displayed. However, interlaced screens induce flicker on some computer screens (use a multi-sync monitor to avoid this):

```
; *** Interlaced Screen example
; *** Filename - InterlacedScreen.bb2

Screen 0,0,0,320,200,3,$0004,"Interlace",1,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.1.4 Extra Half-Brite

Usually known as just "Half-Brite", this is a special display mode which doubles the number of colours on screen by dublicating the existing palette at half its brightness. This doubles the number of available screen colours to 64:

```
; *** Half-Brite Screen example
; *** Filename - HalfBriteScreen.bb2

Screen 0,0,0,320,200,6,$0080,"Half-Brite",1,0
ScreensBitMap 0,0
For A=0 To 31
  Boxf 10,22+X,30,24+X,A
  Boxf 60,22+X,80,24+X,A+32
  Let X+5
Next
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.1.5 Hold And Modify

The Hold And Modify display mode (HAM) uses only 16 colour registers, but manages to display the full Amiga colour palette - all 4096 colours on the screen at the same time. A HAM colour is formed by taking the RGB value of the preceding pixel on the screen, and substituting a new value for one of the RGB components:

```
; *** HAM Screen example
; *** Filename - HAMScreen.bb2

Screen 0,0,0,320,200,6,$0800,"HAM",1,0
ScreensBitMap 0,0
For X=1 To 81
  For A=0 To 50
    Boxf 1+X,12+Y,13+X,17+Y,A
    Let Y+4
  Next
  Let X+9
  Y=0
Next
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.1.6 Screen BitMaps

Screens can also display graphics from a previously initialised BitMap, by the inclusion of the optional BITMAP# parameter. This is of use when the BitMap graphics have been created BEFORE the screen definition.

The SCREENSBITMAP statement can be used to attach BitMap graphics to a screen AFTER it has been opened (consult Chapter 6 for more information):

```
; *** Screen example 2
; *** Filename - Screen2.bb2

; *** Define BitMap (3 bitplanes)
BitMap 0,320,256,3
; *** Draw BitMap graphics
For A=1 To 50
  Circlef Rnd(320),Rnd(150)+50,Rnd(20)+10,Rnd(5)+1
Next A
; *** Open screen and attach BitMap
Screen 0,0,0,320,DispHeight,3,0,"Title",1,0,0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.2 Controlling screens

CLOSESCREEN

```
Mode(s):   Amiga
Statement: close a screen
Syntax:    CloseScreen SCREEN#
```

As its name implies, CLOSESCREEN closes the specified screen and removes it from the display:

```
; *** CloseScreen example
; *** Filename - CloseScreen.bb2

; *** Open screen
Screen 0,1,"Closing down..."
; *** Pause briefly
VWait 50
; *** Remove screen from display
CloseScreen 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.3 Screen priority

As multiple screens are opened, they are positioned in front of one another. The following commands can be used to affect screen priority.

HIDESCREEN

```
Mode(s):   Amiga
Statement: move a screen to back of display
Syntax:    HideScreen SCREEN#
```

The HIDESCREEN statement moves the specified screen to the back of the current display. It places it behind all other opened screens. Try the following example:

```
; *** HideScreen example
; *** Filename - HideScreen.bb2

; *** Open screen
Screen 0,2,"Hide and seek"
; *** Pause briefly
VWait 100
```

```
; *** Move screen to back of display
HideScreen 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SHOWSCREEN

```
Mode(s):   Amiga
Statement: show a screen
Syntax:    ShowScreen SCREEN#
```

SHOWSCREEN moves the specified screen to the front of the current display. For example:

```
; *** ShowScreen example
; *** Filename - ShowScreen.bb2

; *** Open 2 screens
Screen 0,2,"Back"
Screen 1,10,"Front"
For A=1 To 5
  ; *** Pause briefly
  VWait 50
  ; *** Toggle screen priority
  SCR=1-SCR
  ShowScreen SCR
Next A
; *** Return to Blitz Basic 2 editor
End
```

10.4 Manipulating screens

MOVESCREEN

```
Mode(s):   Amiga
Statement: move a screen
Syntax:    MoveScreen SCREEN#,X,Y
```

MOVESCREEN is used to move a screen about the current display. The X and Y parameters specify the amount for the screen to be moved. Here are some examples:

```
; *** MoveScreen example
; *** Filename - MoveScreen.bb2

; *** Open screen
Screen 0,2,"Going down..."
; *** Move screen down display
For Y=1 To 30
  MoveScreen 0,0,Y
Next Y
; *** Return to Blitz Basic 2 editor
End
```

BEEPSCREEN

```
Mode(s):   Amiga
Statement: flash screen
Syntax:    BeepScreen SCREEN#
```

The BEEPSCREEN statement flashes a specified screen (SCREEN#). That's it! Try the following example:

```
; *** BeepScreen example
; *** Filename - BeepScreen.bb2

; *** Open screen
Screen 0,2,"Flasher"
; *** Pause briefly
VWait 50
; *** Flash screen 5 times
For A=1 To 5
  BeepScreen 0
  VWait 50
Next A
; *** Return to Blitz Basic 2 editor
End
```

WBTOSCREEN

```
Mode(s):   Amiga
Statement: assign the Workbench screen to a screen object number
Syntax:    WbToScreen SCREEN#
```

The WBTOSCREEN statement assigns the Workbench screen a screen number. This allows you to perform any of the screen functions on the Workbench screen. Upon execution, the Workbench screen becomes the current screen. For example:

```
; *** WbToScreen example
; *** Filename - WbToScreen.bb2

; *** Assign Workbench screen for manipulation
WbToScreen 0
; *** Pause briefly
VWait 100
; *** Flash screen
BeepScreen 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

FINDSCREEN

```
Mode(s):   Amiga
Statement: assign an object number to a screen
Syntax:    FindScreen SCREEN#[,TITLE$]
```

This statement will search for a screen and give it an object number. If the optional TITLE$ parameter is specified then a screen that has this name will be searched for, otherwise the front screen will be given the object number SCREEN#. If the screen is found then it becomes the current screen. Here is an example:

```
; *** FindScreen!
; *** Filename - FindScreen.bb2

; *** Search for screen 0
FindScreen 0
; *** Open window on found screen
Window 0,0,0,100,100,0,"Found it!",0,1
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SHOWBITMAP

```
Mode(s):   Amiga
Statement: show a BitMap on a screen
Syntax:    ShowBitMap [BITMAP#]
```

The SHOWBITMAP statement is used to display a BitMap on the current screen. It is a system-friendly version of the Slice SHOW statement. This allows the Blitz Basic programmer to create double-buffered animations on Intuition screens. Here is an example:

```
; *** ShowBitmap example
; *** Filename - ShowBitmap.bb2

; *** Open BitMap (3 bitplanes)
BitMap 0,320,DispHeight,3
; *** Plot random starfield on BitMap
For A=1 To 200
  Plot Rnd(320),Rnd(DispHeight),Rnd(5)+1
Next A
; *** Open screen
Screen 0,3
; *** Alter screen's palette
RGB 0,0,0,0
; *** Attach BitMap to screen
ShowBitMap 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SCREENPENS

```
Mode(s):   Amiga
Statement: configure the 10 default pens used for system gadgets
Syntax:    ScreenPens(TEXT,SHINE,SHADOW,FILL,TEXT,BACK,HIGHLIGHT)
```

The SCREENPENS statement is used to configure the 10 default pens used for system gadgets in Workbench 2.0/3.0. All screens opened after the SCREENPENS statement will use these pens. Try the following example which brightens up the dullest of Intuition screens:

```
; *** ScreenPens example
; *** Filename - ScreenPens.bb2

; *** Set screen pens
ScreenPens 1,2,3,4,5,6,7
Screen 0,3,"Hello"
; *** Simple text gadget
TextGadget 0,30,30,0,0,"Click on me"
; *** Open window and attach gadget
Window 0,0,20,300,200,0,"Window",1,2,0
; *** Output defined colours
WLocate 0,6
For A=1 To 7
```

```
  WColour A
  NPrint A
Next A
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.5 Screen functions

SMOUSEX

```
Mode(s):  Amiga
Function: return horizontal mouse position relative to left edge of screen
Syntax:   x=SMouseX
```

The SMOUSEX function returns the horizontal mouse position relative to the left edge of the currently used screen:

```
; *** SMouseX example
; *** Filename - SMouseX.bb2

; *** Open simple screen
Screen 0,3,"Mouse co-ords"
; *** Grab screen's BitMap
ScreensBitMap 0,0
; *** Enable text output onto BitMap
BitMapOutput 0
; *** Alter screen's palette
PalRGB 0,0,0,0,0
Use Palette 0
Repeat
  ; *** Return horizontal location of mouse
  Locate 0,2 : Print "X=",SMouseX,"  "
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

SMOUSEY

```
Mode(s):  Amiga
Function: return vertical mouse position relative to top of screen
Syntax:   y=SMouseY
```

The SMOUSEY function returns the vertical mouse position relative to the top of the currently used screen. For example:

```
; *** SMouseY example
; *** Filename - SMouseY.bb2

; *** Open simple screen
Screen 0,3,"Mouse co-ords"
; *** Grab screen's BitMap
ScreensBitMap 0,0
; *** Enable text output onto BitMap
BitMapOutput 0
; *** Alter screen's palette
PalRGB 0,0,0,0,0
Use Palette 0
Repeat
  ; *** Return vertical location of mouse
  Locate 0,2 : Print "Y=",SMouseY,"  "
Until Joyb(0)>0
; *** Return to Blitz Basic 2 editor
End
```

VIEWPORT

```
Mode(s):  Amiga
Function: return the location of screen's ViewPort
Syntax:   v=ViewPort(SCREEN#)
```

VIEWPORT is used to return the location of a screen's ViewPort. The ViewPort address can be used in conjunction with the Amiga's system libraries:

```
; *** ViewPort Example
; *** Filename - ViewPort.bb2

; *** Use Workbench screen
WbToScreen 0
; *** Output scren's ViewPort
NPrint ViewPort(0)
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.6 IFF screens

IFF stands for Interchangeable File Format. Devised by Electronic Arts, it has been adopted as the standard way of storing pictures and sound on the Amiga.

This section deals with the IFF graphics which can be created with paint packages, such as Deluxe Paint IV.

10.6.1 Loading and saving screens

LOADSCREEN

```
Mode(s):   Amiga
Statement: load a screen into a screen object
Syntax:    LoadScreen SCREEN#,"FILENAME.IFF"[,PALETTE#]
```

LOADSCREEN is used to load an IFF picture ("FILENAME.IFF") into the screen specified by SCREEN#. If the optional PALETTE# parameter is included then the picture's palette will be loaded into that palette object. For example:

```
; *** LoadScreen example
; *** Filename - LoadScreen.bb2

; *** Open screen (32 colours)
Screen 0,5,"Left mouse button exits"
; *** Load screen and palette
LoadScreen 0,"FILENAME.IFF",0
; *** Attach palette to screen
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

SAVESCREEN

```
Mode(s):   Amiga
Statement: save a screen to disk
Syntax:    SaveScreen SCREEN#,"FILENAME.IFF"
```

The SAVESCREEN statement saves a screen (SCREEN#) to disk as an IFF picture file ("FILENAME.IFF"):

```
; *** SaveScreen example
; *** Filename - SaveScreen.bb2

; *** Open screen and attach BitMap
Screen 0,3,"SaveScreen example"
ScreensBitMap 0,0
; *** Alter screen's palette
PalRGB 0,0,0,0,0
Use Palette 0
; *** Plot a random starfield
For A=1 To 100
  Plot Rnd(320),Rnd(200)+30,Rnd(6)+1
Next A
; *** Save IFF file
SaveScreen 0,"df0:STARS.IFF"
Cls 0
VWait 20
; *** Load new file
LoadScreen 0,"df0:STARS.IFF"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.6.2 ILBM

ILBM stands for InterLeaved BitMap - think of it as a posh name for an IFF file. The following commands are used to obtain information about IFF files, including size, number of colours and resolution.

ILBMINFO

```
Mode(s):   Amiga
Statement: initialize a file for ILBM examination
Syntax:    ILBMInfo "FILENAME"
```

Before an ILBM file can be investigated, it first has to be initialized using the ILBMINFO statement. The "FILENAME" parameter is the name of the file to examine:

```
; *** ILBMInfo example
; *** Filename - ILBMInfo.bb2

; *** IFF filename
F$="FILENAME.IFF"
; *** Initialize file
ILBMInfo F$
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

ILBMWIDTH

```
Mode(s):  Amiga
Function: return the width of an ILBM image
Syntax:   w=ILBMWidth
```

This function returns the width of an initialized ILBM image, in pixels:

```
; *** ILBMWidth example
; *** Filename - ILBMWidth.bb2

; *** IFF filename
F$="FILENAME.IFF"
; *** Initialize file
ILBMInfo F$
; *** Output picture's width
NPrint ILBMWidth," pixels"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ILBMHEIGHT

```
Mode(s):  Amiga
Function: return the height of an ILBM image
Syntax:   h=ILBMHeight
```

ILBMHEIGHT returns the hieght of an initialized ILBM image, in pixels:

```
; *** ILBMHeight example
; *** Filename - ILBMHeight.bb2

; *** IFF filename
F$="FILENAME.IFF"
; *** Initialize file
ILBMInfo F$
; *** Output picture's height
NPrint ILBMHeight," pixels"
; *** Wait for a mouse click
MouseWait
```

```
; *** Return to Blitz Basic 2 editor
End
```

ILBMDEPTH

```
Mode(s):  Amiga
Function: return the depth of an ILBM image
Syntax:   d=ILBMDepth
```

The ILBMDEPTH statement returns the depth of an ILBM image, in bitplanes:

```
; *** ILBMDepth example
; *** Filename - ILBMDepth.bb2
; *** IFF filename
F$="FILENAME.IFF"
; *** Initialize file
ILBMInfo F$
; *** Output picture's depth
NPrint ILBMDepth," bitplanes"
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

ILBMVIEWMODE

```
Mode(s):   Amiga/Blitz
Statement: return the viewmode of an ILBM file
Syntax:    ILBMViewMode
```

ILBMViewMode returns the ViewMode, or resolution, of the file that was processed by ILBMInfo. This is useful for opening a screen in the right mode before using LOADSCREEN. The different values of ILBMVIEWMODE are as follows:

Table 10.4 : Values returned by ILBMVIEWMODE

```
Value         Description
========================
32768 ($8000) Hi-res
2048  ($0800) HAM
128   ($0080) Half-Brite
4     ($0004) Interlaced
0     ($0000) Low-res
```

Here is an example:

```
; *** ILBMViewMode example
; *** Filename - ILBMViewMode.bb2

; *** IFF filename
F$="FILENAME.IFF"
; *** Initialize file
ILBMInfo F$
; *** Output picture's ViewMode
NPrint ILBMViewMode
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

Here is a full example which demonstrates the use of the Blitz Basic ILBM commands in the opening of screens:

```
; *** A thorough examination
; *** Filename - ILBM.bb2

; *** Analyse IFF file
F$="FILENAME.IFF"
ILBMInfo F$
; *** Open screen to file specifications
Screen 0,0,0,ILBMWidth,ILBMHeight,ILBMDepth,ILBMViewMode,"",1,2
; *** Load file
LoadScreen 0,F$,0
Use Palette 0
; *** Wait for a mouse click
MouseWait
; *** Return to Blitz Basic 2 editor
End
```

10.7 End-of-Chapter summary

A screen is an area of the display that shares the same attributes, such as size, resolution and colours.

Screen widths must be a multiple of 16 and they are always at least the full width of the viewable area (a minimum of 320 pixels).

The height of a screen will be 256 pixels on a PAL Amiga, or 200 pixels on an NTSC Amiga.

IFF stands for Interchangeable File Format. It has been adopted as the standard way of storing pictures on the Amiga. Blitz Basic can load and save files in this format.

ILBM stands for InterLeaved BitMap. You can obtain information about ILBM files, including size, number of colours and resolution.

Table 10.5 : Screen commands

Command	Function
BEEPSCREEN	Flash a screen
CLOSESCREEN	Close a screen
FINDSCREEN	Search for a screen
HIDESCREEN	Move screen to back of display
ILBMDEPTH	Return depth of IFF image
ILBMHEIGHT	Return height of IFF image
ILBMINFO	Initialize IFF file for examination
ILBMVIEWMODE	Return ViewMode of IFF image
ILBMWIDTH	Return width of IFF image
LOADSCREEN	Load an IFF file
MOVESCREEN	Move a screen
SAVESCREEN	Save an IFF file
SCREEN	Open a screen
SCREENPENS	Set default screen pens
SHOWBITMAP	Display BitMap in a screen
SHOWSCREEN	Move screen to front of display
SMOUSEX	Return x co-ordinate of mouse
SMOUSEY	Return y co-ordinate of mouse
VIEWPORT	Return screen's ViewPort
WBTOSCREEN	Assign screen number to Workbench screen

Chapter 11 : Windows

A window is an independent rectangular area of text and graphics on the screen, which can accept or display information. Windows can be enlarged, shrunk and moved, without altering the main screen. All windows can have a title bar and may contain special gadgets in their borders. (Note that windows must always appear in an Intuition screen).

When using windows the following procedure is recommended:

1. Open a screen using SCREEN or WBTOSCREEN
2. Open a window using WINDOW
3. Use WAITEVENT to detect any user activity in the window
4. Return to step 3

11.1 Opening a window

WINDOW

```
Mode(s):   Amiga
Statement: open an intuition window
Syntax:    Window WINDOW#,X,Y,W,H,F,TITLE$,D,B[,G_LIST#[,BITMAP#]]
```

WINDOW opens the Intuition window index WINDOW#. The X and Y parameters contain the jump coordinates relative to the top left corner of the screen. The W and H parameters contain the width and height of the window.

The F, or FLAGS, parameter specifies the special elements that a window may contain, such as sizing gadgets, close gadgets and drag-bars:

Table 11.1 : The FLAGS parameter

```
Window flag   Value  Description
=========================================================
WINDOWSIZING  $0001  Attach sizing gadget to window
WINDOWDRAG    $0002  Attach drag-bar to window
WINDOWDEPTH   $0004  Attach depth gadget to window
WINDOWCLOSE   $0008  Attach close gadget to window
SIZEBRIGHT    $0010  Leave right hand window margin clear
SIZEBOTTOM    $0020  Leave bottom window margin clear
BACKDROP      $0100  Open window at back of display
GIMMEZEROZERO $0400  Keep border seperate from window area
BORDERLESS    $0800  Open window with no border
ACTIVATE      $1000  Activate the window once opened
```

To use more than one of these flags they must be logically combined using the "|" operator. For example:

```
$0001|$0002|$0004
```

Which, when used as the FLAGS parameter, would attach a sizing gadget, drag-bar and depth gadget to the window.

TITLE$ is a string which contains the title of the window, to be displayed at the very top of the window. If you do not want a title for the window then use a null string for TITLE$ ("").

The D parameter specifies the colour of the detail pen of the window, as used in the window title. B is the block pen of the window, as used in the window border.

The optional G_LIST# parameter is the number of a gadgetlist object to be attached to the window - consult Chapter 13 for more information.

Here are some examples:

```
; *** Window examples
; *** Filename - Window.bb2

WbToScreen 0
WBenchToFront_
Window 0,0,0,150,100,$0001,"Sizing gadget",1,0
Window 1,150,0,150,100,$0002,"Drag gadget",1,0
Window 2,300,0,150,100,$0004,"Depth gadget",1,0
Window 3,450,0,150,100,$0008,"Close gadget",1,0
Window 4,0,100,150,100,$0001|$0008,"Sizing & Close",1,0
MouseWait
WBenchToBack_
End
```

11.1.1 Super-BitMap windows

Super-BitMap windows can also be created. These allow the window to have its own BitMap which can be physically larger than the window. The BitMap can then be scrolled about the window. To attach a BitMap to a window, set the SuperBitMap flag in the FLAGS parameter and include the number of the BitMap to be attached in the BITMAP# parameter in your window definition.

GETSUPERBITMAP

```
Mode(s):   Amiga
Statement: get super-BitMap
Syntax:    GetSuperBitMap
```

PUTSUPERBITMAP

```
Mode(s):   Amiga
Statement: put super-BitMap
Syntax:    PutSuperBitMap
```

GETSUPERBITMAP is used to grab the contents of a super-BitMap. This allows you to update the contents of the super-BitMap.

PUTSUPERBITMAP is used to put (i.e. refresh) the super-BitMap back into the current window.

In the following example, which demonstrates the above commands, try pressing the left mouse button in the window to clear the super-BitMap:

```
; *** GetSuperBitMap/PutSuperBitMap example
; *** Filename - PutSuperBitMap.bb2

BitMap 0,320,256,3
; *** Draw BitMap graphics
Boxf 0,0,319,255,4
For A=1 To 7
  Circlef 160,100,160-A*5,100-A*5,A
Next A
Screen 0,11,"My Screen"
WIDTH=320
HEIGHT=256
PropGadget 0,3,-8,$18000+4+8+64,1,-20,8
PropGadget 0,-14,10,$11000+2+16+128,2,12,-20
AddIDCMP $10
SizeLimits 32,32,320+22,256+20
Window 0,0,20,200,150,$1489,"Window",1,2,0,0
Gosub DRAW

; *** Main loop
Repeat
  ev.l=WaitEvent
  If ev=2 Then Gosub SIZE
  If ev=$8 Then Gosub ALTER
  If ev=$20 Then Gosub MOVIE
Until ev=$200
End

; *** Draw sliders
SIZE:
SetHProp 0,1,X/WIDTH,InnerWidth/WIDTH
SetVProp 0,2,Y/HEIGHT,InnerHeight/HEIGHT
Redraw 0,1
Redraw 0,2
Goto DRAW
```

```
; *** Move SuperBitMap
MOVIE:
Repeat
Gosub DRAW
Until WaitEvent<>$10
Return

; *** Position SuperBitMap
DRAW:
W=WIDTH-InnerWidth
H=HEIGHT-InnerHeight
X=QLimit(HPropPot(0,1)*(W+1),0,W)
Y=QLimit(VPropPot(0,2)*(H+1),0,H)
PositionSuperBitMap X,Y
Return

; *** Alter the contents of SuperBitMap
ALTER:
GetSuperBitMap
Cls
PutSuperBitMap
Return
```

POSITIONSUPERBITMAP

```
Mode(s):   Amiga
Statement: position a super-BitMap
Syntax:    PositionSuperBitMap X,Y
```

This statement is used to position the BitMap in the current super-BitMap window. The X and Y parameters specify the new co-ordinates of the top left-hand corner of the BitMap. Here's an example:

```
; *** PositionSuperBitMap example
; *** Filename - PositionSuperBitMap.bb2

BitMap 0,320,256,3
; *** Draw BitMap graphics
Boxf 0,0,319,255,4
For A=1 To 7
  Circlef 160,100,160-A*5,100-A*5,A
Next A
Screen 0,11,"My Screen"
WIDTH=320
HEIGHT=256
PropGadget 0,3,-8,$18000+4+8+64,1,-20,8
PropGadget 0,-14,10,$11000+2+16+128,2,12,-20
AddIDCMP $10
SizeLimits 32,32,320+22,256+20
```

```
Window 0,0,20,200,150,$1489,"Window",1,2,0,0
Gosub DRAW

; *** Main loop
Repeat
  ev.l=WaitEvent
  If ev=2 Then Gosub SIZE
  If ev=$20 Then Gosub MOVIE
Until ev=$200
End

; *** Draw sliders
SIZE:
SetHProp 0,1,X/WIDTH,InnerWidth/WIDTH
SetVProp 0,2,Y/HEIGHT,InnerHeight/HEIGHT
Redraw 0,1
Redraw 0,2
Goto DRAW

; *** Move SuperBitMap
MOVIE:
Repeat
Gosub DRAW
Until WaitEvent<>$10
Return

; *** Position SuperBitMap
DRAW:
W=WIDTH-InnerWidth
H=HEIGHT-InnerHeight
X=QLimit(HPropPot(0,1)*(W+1),0,W)
Y=QLimit(VPropPot(0,2)*(H+1),0,H)
PositionSuperBitMap X,Y
Return
```

11.2 Manipulating windows

The following commands are used to manipulate windows after they have been created with the WINDOW statement.

11.2.1 Moving between windows

USE WINDOW

```
Mode(s):   Amiga
Statement: set current window
Syntax:    Use Window WINDOW#
```

The USE WINDOW statement sets the specified window (WINDOW#) as the currently used window. USE WINDOW automatically performs a WINDOWINPUT and WINDOWOUTPUT on the window. For example:

```
; *** Use Window example
; *** Filename - Use Window.bb2

WbToScreen 0
WBenchToFront_
Window 0,0,10,300,100,$0001,"",1,0
Window 1,300,10,300,100,$0001,"",1,0
Use Window 0
Print "Hello from window 0"
Use Window 1
Print "Hello from window 1"
MouseWait
WBenchToBack_
End
```

11.2.2 Closing a window

FREE WINDOW

```
Mode(s):   Amiga
Statement: close a window
Syntax:    Free Window WINDOW#
```

FREE WINDOW closes the specified window and removes it from the display. Here is an example:

```
; *** Free Window example
; *** Filename - Free Window.bb2

WbToScreen 0
WBenchToFront_
Window 0,0,10,300,100,$0001,"Click mouse",1,0
MouseWait
Free Window 0
VWait 50
End
```

CLOSEWINDOW

```
Mode(s):   Amiga
Statement: close a window
Syntax:    CloseWindow WINDOW#

CLOSEWINDOW works exactly the same as FREE WINDOW. Why? Who knows? Just for
the sake of it, here's the same example, but this time using CLOSEWINDOW:

; *** CloseWindow example
; *** Filename - CloseWindow.bb2

WbToScreen 0
WBenchToFront_
Window 0,0,10,300,100,$0001,"Click mouse",1,0
MouseWait
CloseWindow 0
VWait 50
End
```

11.2.3 Activating a window

ACTIVATE

```
Mode(s):   Amiga
Statement: activate a window
Syntax:    Activate WINDOW#
```

The ACTIVATE statement is used to activate a specified window (WINDOW#). For example:

```
; *** Activate example ** Filename - Activate.bb2 **

Screen 0,2
Window 0,0,0,320,100,0,"Window 1",0,1
Window 1,0,100,320,100,0,"Window 2",0,1
VWait 100
Activate 0
Use Window 0
Print "Hello from Window 1"
VWait 100
Activate 1
Use Window 1
Print "Hello from Window 2"
MouseWait
End
```

11.2.4 Window titles

WTITLE

```
Mode(s):   Amiga
Statement: update window and screen title
Syntax:    WTitle "WINDOW TITLE","SCREEN TITLE"
```

The WTITLE statement is used to alter, or update, the current window and screen titles. For example:

```
; *** WTitle example
; *** Filename - WTitle.bb2

Screen 0,3,"Blitz"
Window 0,0,12,320,200,0,"Basic",1,2
Repeat
  ev.l=WaitEvent
Until ev=$8
WTitle "Tops","Is"
VWait 20
MouseWait
End
```

11.2.5 Altering window menus

MENUS

```
Mode(s):   Amiga
Statement: turn ALL menus on or off
Syntax:    Menus On/Off
```

The MENUS statement may be used to turn ALL menus on or off in the currently used window. Here is an example:

```
; *** Menus example
; *** Filename - Menus.bb2

MenuTitle 0,0,"PROJECT"
MenuItem 0,0,0,0,"Load"
Screen 0,3
Window 0,0,20,320,100,$0001+$0008+$100f,"",1,2
SetMenu 0
; *** Press left mouse to toggle menus on/off
T=1
Repeat
```

```
    ev.l=WaitEvent
    If Joyb(0)=1
      If T=1
        Menus Off
      Else
        Menus On
      End If
      T=1-T
    EndIf
  Until ev=$200
  End
```

11.2.6 Moving a window

WMOVE

```
Mode(s):   Amiga
Statement: move the current window
Syntax:    WMove X,Y
```

This statement physically moves the currently used window to the coordinates specified by the X and Y parameters. For example:

```
; *** WMove examples
; *** Filename - WMove.bb2

Screen 0,2
Window 0,0,0,150,100,0,"Moving window",0,1
VWait 100
WMove 100,0
VWait 50
WMove 100,100
MouseWait
End
```

11.2.7 Window scrolling

WSCROLL

```
Mode(s):   Amiga
Statement: scroll a rectangular area of the current window
Syntax:    WScroll X1,Y1,X2,Y2,DELTA_X,DELTA_Y
```

The WSCROLL statement is used to scroll a portion of the current window. X1 and Y1 are the coordinates of the top left-hand corner or the area to scroll and X2 and Y2 are the bottom right-hand coordintes. The DELTA_X and DELTA_Y parameters specify the amount the area is to be moved, in the following directions:

Table 11.2 : Delta values

```
Delta values  DELTA_X  DELTA_Y
==============================
Positive      Left     Up
Negative      Right    Down
```

Here's an example:

```
; *** WScroll example
; *** Filename - WScroll.bb2

Screen 0,3
Window 0,0,20,320,200,0,"Scrolled",0,1
For A=1 To 50
  WCircle Rnd(260)+30,Rnd(100)+40,Rnd(20)+5,Rnd(8)
Next A
VWait 50
WScroll 4,13,310,180,0,-20
VWait 50
WScroll 4,13,310,180,0,20
MouseWait
End
```

11.2.8 Window sizing

WSIZE

```
Mode(s):   Amiga
Statement: alter the width and height of the current window
Syntax:    WSize WIDTH,HEIGHT
```

WSIZE is used to change the size of the currently used window. WIDTH and HEIGHT are measured in pixels:

```
; *** WSize example
; *** Filename - WSize.bb2

Screen 0,2
Window 0,0,0,150,100,0,"Growing window",0,1
VWait 100
WSize 320,200
```

```
MouseWait
End
```

SIZELIMITS

```
Mode(s):   Amiga
Statement: set the limits that windows can be sized with sizing gadget
Syntax:    SizeLimits MIN_WIDTH,MIN_HEIGHT,MAX_WIDTH,MAX_HEIGHT
```

SIZELIMITS is used to set the limits that any new windows can be sized to using the sizing gadget. The MIN_WIDTH and MIN_HEIGHT parameters define the minimum size of the window, and MAX_WIDTH and MAX_HEIGHT the maximum size, in pixels. Try the following example:

```
; *** Size restrictions
; *** Filename - SizeLimits.bb2

Screen 0,2
SizeLimits 100,100,320,100
Window 0,0,00,320,100,$0001+$0008,"Change my size",0,1
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

11.2.9 Window BitMaps

BITMAPTOWINDOW

```
Mode(s):   Amiga
Statement: copy a BitMap to a window
Syntax:    BitmapToWindow BITMAP#,WINDOW#[,X1,Y1,X2,Y2,W,H]
```

This statement is used to copy a BitMap (BITMAP#) to a window (WINDOW#). The optional parameters are as follows:

Table 11.3 : BITMAPTOWINDOW parameters

```
Parameter  Description
===============================================
X1         X co-ordinate of BitMap
Y1         Y co-ordinate of BitMap
X2         X co-ordinate of window
Y2         Y co-ordinate of window
```

```
W          Width of BitMap to copy (in pixels)
H          Height of BitMap to copy (in pixels)
```

For example:

```
; *** BitMaptoWindow example
; *** Filename - BitMaptoWindow.bb2

Screen 0,3
ScreensBitMap 0,0
Cls
For A=1 To 100
  Circlef Rnd(320),Rnd(80),Rnd(20)+10,Rnd(5)+1
Next A
Window 0,0,110,320,100,0,"Click mouse button",1,2
BitMaptoWindow 0,0,20,20,10,15,300,80
Repeat
  ev.l=WaitEvent
Until ev=$8
End
```

11.3 Window functions

The following functions return information about previously initalised windows.

11.3.1 Window dimensions

WINDOWX

```
Mode(s):  Amiga
Function: return horizontal location of the top left corner of the window
Syntax:   x=WindowX
```

This function returns the horizontal location, in pixels, of the top left-hand corner of the currently used window, relative to the screen that the window appears in.

To return the vertical location of the window, use the corresponding WINDOWY function.

WINDOWY

```
Mode(s):  Amiga
Function: return the vertical location of the top left corner of the window
Syntax:   y=WindowY
```

For example:

```
; *** WindowX/Y example
; *** Filename - WindowY.bb2

Screen 0,2
Window 0,Rnd(200)+10,Rnd(100)+10,150,100,0,"",0,1
WLocate 0,0
NPrint "Window X = ",WindowX
NPrint "Window Y = ",WindowY
MouseWait
End
```

WINDOWWIDTH

```
Mode(s):  Amiga
Function: return the width of the current window
Syntax:   w=WindowWidth
```

WINDOWWIDTH returns the width of the currently used window.

WINDOWHEIGHT

```
Mode(s):  Amiga
Function: return the height of the current window
Syntax:   h=WindowHeight
```

WINDOWHEIGHT returns the height of the currently used window:

```
; *** Window dimensions
; *** Filename - WindowHeight.bb2

Screen 0,2
Window 0,0,20,Rnd(150)+170,Rnd(100)+10,0,"",0,1
WLocate 0,0
NPrint "Window width = ",WindowWidth
NPrint "Window height = ",WindowHeight
MouseWait
End
```

INNERWIDTH

```
Mode(s):  Amiga
Function: return the width inside the border of the current window
Syntax:   w=InnerWidth
```

The INNERWIDTH function returns the width, in pixels, of the area inside the border of the currently used window.

INNERHEIGHT

```
Mode(s):  Amiga
Function: return the height inside the border of the current window
Syntax:   h=InnerHeight
```

INNERHEIGHT returns the height, in pixels, of the area inside the border of the currently used window. For example:

```
; *** Window dimensions 2
; *** Filename - InnerHeight.bb2

Screen 0,2
Window 0,0,20,Rnd(150)+170,Rnd(100)+10,0,"",0,1
WLocate 0,0
NPrint "Inner width = ",InnerWidth
NPrint "Inner height = ",InnerHeight
MouseWait
End
```

WTOPOFF

```
Mode(s):  Amiga
Function: return distance between top of window border and its inside
Syntax:   t=WTopOff
```

This function returns the distance between the top of the current window border and the inside of the window, in pixels.

WLEFTOFF

```
Mode(s):  Amiga
Function: return distance between left edge of window border and its inside
Syntax:   l=WLeftOff
```

The WLEFTOFF function returns the distance between the left edge of the current window border and the inside of the window, in pixels:

```
; *** Window dimensions 3
; *** Filename - WLeftOff.bb2

Screen 0,2
Window 0,0,20,320,100,0,"Window",0,1
WLocate 0,0
NPrint "WTopOff = ",WTopOff
NPrint "WLeftOff = ",WLeftOff
MouseWait
End
```

11.3.2 Window RastPort

RASTPORT

```
Mode(s):  Amiga
Function: return the specified Window's RastPort address
Syntax:   r=RastPort(WINDOW#)
```

This function returns the RastPort address of the specified window:

```
; *** RastPort example
; *** Filename - RastPort.bb2

Screen 0,2
Window 0,0,00,320,100,$0001+$0008,"",0,1
NPrint "RastPort = ",RastPort(0)
MouseWait
End
```

11.4 Window events

11.4.1 IDCMP flags

IDCMP flags are special flags which are attached to windows. They describe the type of "event" which can be reported by a window. Events occur when a window has its size changed with the sizing gadgets, or when a mouse button is pressed, or when a disk is removed etc. (see below for full list). Events are reported by the WAITEVENT and EVENT functions.

Table 11.4 : IDCMP flags

```
IDCMP flag  Event
=====================================================
$2          Reported when window has its size changed
$4          Reported when window contents corrupted
$8          Reported when mouse button is pressed
$10         Reported when mouse has been moved
$20         Reported when window gadget has been pushed down
$40         Reported when window gadget has been released
$100        Reported when menu operation in window has occured
$200        Reported when close gadget has been selected
$400        Reported upon keypress
$8000       Reported when disk is inserted
$10000      Reported when disk is removed
$40000      Reported when window has been activated
$80000      Reported when window has been de-activated
```

11.4.2 Defining IDCMP flags

By default, all windows are opened with an IDCMP flags setting of:

```
$02|$4|$8|$20|$40|$100|$200|$400|$40000|$80000
```

However, this may be changed by the DEFAULTIDCMP statement.

DEFAULTIDCMP

```
Mode(s):   Amiga
Statement: set window IDCMP flags
Syntax:    DefaultIDCMP IDCMP_FLAGS
```

This statement is used to define the window IDCMP flags. Each window can have its own set of IDCMP flags, although they must be defined before the window is opened.

If you require more than one IDCMP flag then you can use the (|) operator to add them together. Here are some examples:

```
; *** DefaultIDCMP example
; *** Filename - DefaultIDCMP.bb2

Screen 0,3
; *** This example closes a window using
; *** the mouse button, not a close gagdet
DefaultIDCMP $8
Window 0,0,20,320,100,0,"Press mouse button",0,1
ev.l=WaitEvent
If ev=$8 Then Free Window 0
VWait 100
End
```

```
; *** DefaultIDCMP example 2
; *** Filename - DefaultIDCMP2.bb2

Screen 0,3
; *** This example flashes the screen
; *** upon a key-press
DefaultIDCMP $8|$400
Window 0,0,20,320,100,$1000,"Press a key",0,1
Repeat
  ev.l=WaitEvent
  If ev=$400 Then BeepScreen 0
Until ev=$8
End
```

```
; *** DefaultIDCMP example 3
; *** Filename - DefaultIDCMP3.bb2

Screen 0,3
; *** This example ends when the
; *** close gadget is selected
DefaultIDCMP $200
Window 0,0,20,320,100,$0008|$1000,"Hit close gadget",0,1
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

```
; *** DefaultIDCMP example 4
; *** Filename - DefaultIDCMP4.bb2

Screen 0,3
; *** This example ends when the mouse pointer
; *** is moved
DefaultIDCMP $10
Window 0,0,20,320,100,$1000,"Move mouse pointer",0,1
Repeat
  ev.l=WaitEvent
Until ev=$10
End
```

```
; *** DefaultIDCMP example 5
; *** Filename - DefaultIDCMP5.bb2

Screen 0,3
; *** This example ends when a disk is
; *** is inserted into a disk drive
DefaultIDCMP $80000
Window 0,0,20,320,100,$1000,"Insert a disk",0,1
Repeat
  ev.l=WaitEvent
Until ev=$80000
End
```

11.4.3 Adding IDCMP flags

ADDIDCMP

```
Mode(s):   Amiga/Blitz
Statement: add IDCMP flags
Syntax:    AddIDCMP IDCMP_FLAGS
```

This statement is used to add IDCMP flags to those selected by the DEFAULTIDCMP statement. ADDIDCMP must be executed before the window is opened. Here are some examples:

```
; *** AddIDCMP example
; *** Filename - AddIDCMP.bb2

Screen 0,3
; *** Close gadget exits program
DefaultIDCMP $200
; *** Add IDCMP flag (key-press)
AddIDCMP $400
Window 0,0,20,320,100,$0008|$1000,"Close me",0,1
Repeat
  ev.l=WaitEvent
  ; *** Flash screen upon key-press
  If ev=$400 Then BeepScreen 0
```

```
Until ev=$200
End
```

```
; *** AddIDCMP example 2
; *** Filename - AddIDCMP2.bb2

Screen 0,3
; *** Close gadget exits program
DefaultIDCMP $200
; *** Add IDCMP flag (window de-activation)
AddIDCMP $80000
Window 0,0,20,320,100,$0008|$1000,"Close me",0,1
Repeat
  ev.l=WaitEvent
  ; *** Clear window upon de-activation
  If ev=$80000 Then InnerCls Rnd(4)+1 : Activate 0
Until ev=$200
End
```

11.4.4 Subtracting IDCMP flags

SUBIDCMP

```
Mode(s):   Amiga/Blitz
Statement: subtract IDCMP flags
Syntax:    SubIDCMP IDCMP_FLAGS
```

Similarly, the SUBIDCMP statement subtracts IDCMP flags from those selected by the DEFAULTIDCMP statement. SUBIDCMP must be executed before the window is opened. Examples:

```
; *** SubIDCMP example
; *** Filename - SubIDCMP.bb2

Screen 0,3
; *** Close gadget exits program
DefaultIDCMP $200|$80000
; *** Remove de-activation IDCMP flag
SubIDCMP $80000
Window 0,0,20,320,100,$0008|$1000,"Close me",0,1
Repeat
  ev.l=WaitEvent
  ; *** This line won't work
  If ev=$80000 Then InnerCls Rnd(4)+1 : Activate 0
Until ev=$200
End
```

```
; *** SubIDCMP example 2
; *** Filename - SubIDCMP2.bb2

Screen 0,3
; *** Close gadget exits program
DefaultIDCMP $200|$400
; *** Try removing the following line
SubIDCMP $400
Window 0,0,20,320,100,$0008|$1000,"Close me",0,1
Repeat
  ev.l=WaitEvent
  ; *** This line won't work
  If ev=$400 Then End
Until ev=$200
End
```

11.4.5 Window event functions

The following commands are used to return the IDCMP flags of the Intuition events which occur in windows.

WAITEVENT

```
Mode(s): Amiga
Command: return the IDCMP flag of an Intuition event
Syntax:  ev.l=WaitEvent
```

WAITEVENT, as a statement, is used to halt program execution until an Intuition event occurs. This event must satisfy the relevant IDCMP flags.

Used as a function, WAITEVENT will return the IDCMP flag of the event. If no event has occured then (0) is returned.

Note that the value returned by the WAITEVENT function must be assigned to a long type variable (e.g. ev.l=WaitEvent). For example:

```
; *** WaitEvent example
; *** Filename - WaitEvent.bb2

Screen 0,3
Window 0,0,20,320,100,$0008,"Close me",0,1
; *** Exit only if close gadget is selected
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

EVENT

```
Mode(s):  Amiga
Function: return the IDCMP flag of an Intuition event
Syntax:   ev.l=Event
```

The EVENT function also returns the IDCMP flag of an Intuition event, except it does not halt program execution. If no event has occured then (0) is returned. Here's an example:

```
; *** Event example
; *** Filename - Event.bb2

Screen 0,3
ScreensBitMap 0,0
DefaultIDCMP $400
Window 0,0,20,320,200,$1000,"Press any key",0,1
While Event=0
  InnerCls Rnd(7)+1
  VWait 6
Wend
End
```

EVENTWINDOW

```
Mode(s):  Amiga/Blitz
Function: return the window number in which a window event occured
Syntax:   e=EventWindow
```

This statement returns the number of the window in which the most recent window event occured. Window events are those detected by the WAITEVENT or EVENT statements. For example:

```
; *** EventWindow example
; *** Filename - EventWindow.bb2

Screen 0,3
Window 0,0,0,160,100,$100f,"Window 0",1,2
Window 1,160,0,160,100,$100f,"Window 1",1,2
Window 2,0,100,160,100,$100f,"Window 2",1,2
Window 3,160,100,160,100,$100f,"Window 3",1,2
; *** Try fiddling with the windows in this demo
; *** (Press escape to quit)
Repeat
  ev.l=WaitEvent
  Use Window lw
  InnerCls
  Use Window EventWindow
  WLocate 0,0
  Print "Event here!"
  lw=EventWindow
Until Inkey$=Chr$(27)
End
```

11.4.6 Gadget events

GADGETHIT

```
Mode(s):  Amiga
Function: return number of gadget that caused the last gadget event
Syntax:   g=GadgetHit
```

The GADGETHIT function returns the number of the gadget that caused the last gadget event. As gadgets in different windows may use the same identification numbers, the EVENTWINDOW statement may be used to locate the correct gadget. If no gadgets have been selected then (-1) will be returned. For example:

```
; *** GadgetHit example
; *** Filename - GadgetHit.bb2

Screen 0,3
TextGadget 0,20,20,0,1,"A gadget"
TextGadget 0,20,40,0,2,"Another gadget"
TextGadget 0,20,60,0,3,"QUIT"
Window 0,0,0,160,100,0,"A Window",1,2,0
Repeat
  Repeat
    ev.l=WaitEvent
  Until ev=$40
  If GadgetHit=3 Then End
Forever
```

11.4.7 Menu events

MENUHIT

```
Mode(s):  Amiga
Function: return number of menu that caused the last menu event
Syntax:   m=MenuHit
```

MENUHIT returns the number of the menu that caused the last menu event. As menus in different windows may use the same identification numbers, the EVENTWINDOW statement may be used to locate the correct menu. If no menus have been selected then (-1) will be returned:

```
; *** MenuHit example
; *** Filename - MenuHit.bb2

Screen 0,3
Window 0,0,20,160,100,$1000,"Press right mouse",1,2
MenuColour 5
MenuTitle 0,0,"This is a menu title"
MenuItem 0,0,0,0,"This is a menu item"
MenuItem 0,0,0,1,"And this is another"
SetMenu 0
While MenuHit<>0
  ev.l=WaitEvent
Wend
End
```

ITEMHIT

```
Mode(s):  Amiga
Function: return number of menu item that caused the last menu event
Syntax:   i=ItemHit
```

This function returns the number of the menu item that caused the last menu event. If no items have been selected then (-1) will be returned. For example:

```
; *** ItemHit example
; *** Filename - ItemHit.bb2

Screen 0,3
Window 0,0,20,160,100,$1000,"Press right mouse",1,2
MenuColour 5
MenuTitle 0,0,"Project"
MenuItem 0,0,0,0,"Load"
MenuItem 0,0,0,1,"QUIT"
SetMenu 0
Repeat
  WaitEvent
Until ItemHit=1
End
```

SUBHIT

```
Mode(s):  Amiga
Function: return number of menu subitem that caused the last menu event
Syntax:   s=SubHit
```

The SUBHIT function returns the number of the menu subitem that caused the last menu event. If no subitem has been selected then (-1) will be returned. Here is an example:

```
; *** SubHit example
; *** Filename - SubHit.bb2

Screen 0,3
Window 0,0,20,160,100,$1000,"Press right mouse",1,2
MenuColour 5
MenuTitle 0,0,"Project"
MenuItem 0,0,0,0,"Hello"
SubItem 0,0,0,0,0,"QUIT"
SetMenu 0
Repeat
  WaitEvent
Until SubHit=0
End
```

11.4.8 Keyboard events

EVENTCODE

```
Mode(s):  Amiga
Function: return keyboard event code
Syntax:   e=EventCode
```

This function returns the code of the last keyboard event. For example:

```
; *** EventCode example
; *** Filename - EventCode.bb2

Screen 0,3
Window 0,0,0,320,200,$1000,"Amiga mode",0,1
Repeat
  ev.l=WaitEvent
  A$=Inkey$
Until ev.l>0
NPrint EventCode
```

```
VWait 50
End
```

EVENTQUALIFIER

```
Mode(s):  Amiga
Function: return keyboard event qualifier
Syntax:   e=EventQualifier
```

The EVENTQUALIFIER function returns a code which indicates the control key used during the most recent keyboard event. Below is a table of of possible codes:

Table 11.5 : Qualifier codes

Control key	Code
[None]	$8000
[Ctrl]	$8008
[Caps Lock]	$8004
Left [Shift]	$8001
Right [Shift]	$8002
Left [Alt]	$8010
Right [Alt]	$8020
Left [Amiga]	$8040
Right [Amiga]	$8080

Try this example:

```
; *** EventQualifier example
; *** Filename - EventQualifier.bb2

Screen 0,3
DefaultIDCMP $400
Window 0,0,20,320,100,$1000,"Type something",0,1
Repeat
  ev.l=WaitEvent
Until ev=$400
WLocate 0,0
NPrint Hex$(EventQualifier)
VWait 80
End
```

11.4.9 Clearing the event queue

FLUSHEVENTS

```
Mode(s):   Amiga/Blitz
Statement: clear events from event queue
Syntax:    FlushEvents [IDCMP_FLAGS]
```

Window events are automatically stored in a special event queue, or storage buffer, so that they can be read at a later stage in your program. FLUSHEVENTS can be used to clear the event queue, if necessary. If the optional IDCMP_FLAGS parameter is included then only these events are cleared.

11.5 Window text

All Workbench-based applications use text to a certain degree. Text can be used to prompt the user for input, or to display a message, or help file.

As already explained, the PRINT and NPRINT statements are used to output text onto BitMaps, and in windows. Hence the following commands are provided by Blitz Basic for the manipulation of window text.

WINDOWOUTPUT

```
Mode(s):   Amiga/Blitz
Statement: cause print commands to output to window object
Syntax:    WindowOutput WINDOW#
```

The WINDOWOUTPUT statement causes all future print statements (PRINT and NPRINT) to send their output to the specified window. WINDOWOUTPUT is automatically executed when a window is opened or USE WINDOW is executed. When a window is closed, WINDOWOUTPUT must be used to re-direct print output. For example:

```
; *** WindowOutput example
; *** Filename - WindowOutput.bb2

Screen 0,3
Window 0,0,0,320,100,0,"",0,1
Window 1,0,100,320,100,0,"",0,1
WindowInput 0
WindowOutput 0
Activate 0
A$=Edit$("Type something",15)
End
```

WPRINTSCROLL

```
Mode(s):   Amiga
Statement: force text to be scrolled in window
Syntax:    WPrintScroll
```

This statement is used to scroll the current window contents upwards if the text cursor reaches the bottom of the window. WPRINTSCROLL only works with windows opened with flag ($400) set. For example:

```
; *** WPrintScroll example
; *** Filename - WPrintScroll.bb2

Screen 0,3
Window 0,0,15,320,100,$400,"",1,2
For A=0 To 200
  VWait
  NPrint "Scrolling ",A
  WPrintScroll
Next A
End
```

11.5.1 Changing the text style

LOADFONT

```
Mode(s):   Amiga
Statement: load an intuition font
Syntax:    LoadFont FONT#,"FILENAME.FONT",Y_SIZE[,STYLE]
```

WINDOWFONT

```
Mode(s):   Amiga
Statement: set intuition font for current window
Syntax:    WindowFont FONT#
```

The LOADFONT statement is used to load an intuition font into memory. The Y_SIZE parameter is the size of the intuition font to load. If the optional STYLE parameter is included then styling may be applied to the font.

Table 11.6 : The STYLE parameter

```
STYLE  Description
=====================
1      Underlined
2      Bold
4      Italic
8      Width increase
```

WINDOWFONT is used to set the font for the currently used window. Any further text ouput to this window will appear in this text style. The FONT# parameter specifies a previously initialised intuition font. For example:

```
; *** LoadFont/WindowFont example
; *** Filename - LoadFont.bb2

Screen 0,3,"Hello"
Window 0,0,20,320,200,$1000,"Fonts",1,0
NPrint "Normal"
LoadFont 0,"FILENAME.FONT",12
WindowFont 0
NPrint "Your font"
MouseWait
End
```

11.5.2 Setting the text colour

WCOLOUR

```
Mode(s):   Amiga
Statement: set foreground and background colour of window text
Syntax:    WColour FOREGROUND,BACKGROUND
```

The WCOLOUR statement specifies the foreground and background colour of all text printed in the currently used window. For example:

```
; *** A splash of WColour
; *** Filename - WColour.bb2

Screen 0,3
Window 0,0,20,320,200,$1000,"Text colour",1,0
WColour 0,1
NPrint "Hilight"
WColour 1,0
```

```
NPrint "No hilight"
MouseWait
End
```

11.5.3 Changing the text mode

WJAM

```
Mode(s):   Amiga
Statement: set window text drawing mode
Syntax:    WJam MODE
```

WJAM is used to select which parts of the text characters are printed on the currently used window. MODE is a special parameter which sets the print mode:

Table 11.7 : WJam modes

Value	Mode	Description
0	Jam1	Only the foreground colour to be printed
1	Jam2	Print foreground and background colours
2	Complement	Combine characters (XOR)
4	Inverse	Print inverse video characters

Here is an example:

```
; *** WJam session ** Filename - WJam.bb2

Screen 0,3
Window 0,0,20,320,200,0,"Examples",1,0
Print "Overlap"
VWait 100
WJam 0
WLocate 0,0
Print "Rubbish"
VWait 100
WJam 1
WLocate 0,0
Print "Rubbish"
VWait 100
WLocate 0,10
WJam 4
Print "Inverse video"
MouseWait
End
```

11.5.4 The text cursor

WLOCATE

```
Mode(s):   Amiga/Blitz
Statement: set the text cursor position
Syntax:    WLocate X,Y
```

The WLOCATE statement sets the text cursor position in the currently used window. The X and Y parameters specify the distance, in pixels, from the upper left corner of the window. Each window can have a different text cursor position. For example:

```
; *** WLocate example
; *** Filename - WLocate.bb2

Screen 0,2
Window 0,0,20,170,100,0,"",0,1
WLocate 0,0
NPrint "Top left"
WLocate 60,85
NPrint "Bottom right"
MouseWait
End
```

The following two functions return the horizontal and vertical positions of the text cursor respectively.

WCURSX

```
Mode(s):  Amiga
Function: return the horizontal cursor position
Syntax:   x=WCursX
```

For example:

```
; *** WCursX example ** Filename - WCursX.bb2

Screen 0,2
Window 0,0,20,170,100,0,"Cursor X =",0,1
For A=1 To 5
  Print WCursX
VWait
Next A
MouseWait
End
```

WCURSY

```
Mode(s):  Amiga
Function: return the vertical cursor position
Syntax:   y=WCursY
```

Here's an example:

```
; *** WCursY example
; *** Filename - WCursY.bb2

Screen 0,2
Window 0,0,20,170,100,0,"Cursor Y =",0,1
For A=1 To 5
  NPrint WCursY
VWait
Next A
MouseWait
End
```

11.6 Window input

WINDOWINPUT

```
Mode(s):   Amiga/Blitz
Statement: cause input commands to receive info from window object
Syntax:    WindowInput WINDOW#
```

The WINDOWINPUT statement causes all future input functions to receive their input from the specified window (WINDOW#). WINDOWINPUT is automatically executed when a window is opened or USE WINDOW is executed. When a window is closed, WINDOWINPUT must be used to re-direct keyboard input. Try this example:

```
; *** WindowInput example
; *** Filename - WindowInput.bb2

Screen 0,3
Window 0,0,0,320,100,0,"",0,1
Window 1,0,100,320,100,0,"",0,1
WindowInput 0
WindowOutput 0
Activate 0
A$=Edit$("Type something",15)
End
```

11.6.1 Reading the keyboard

RAWKEY and QUALIFIER are used to read the 96 alphanumeric and special keys which make up the Amiga's keyboard.

RAWKEY

```
Mode(s):  Amiga
Function: return the raw key code of the most recent key-press
Syntax:   k=RawKey
```

RAWKEY returns the code of a key that has already been detected using the INKEY$ function. Here is an example:

```
; *** RawKey example
; *** Filename - RawKey.bb2

Screen 0,3
Window 0,0,0,320,200,$1000,"Hello!",0,1
Repeat
  ev.l=WaitEvent
  WLocate 0,0
  A$=Inkey$
  Print RawKey
Until ev.l=$8
End
```

QUALIFIER

```
Mode(s):  Amiga
Function: return the control key(s) used in the most recent key-press
Syntax:   q=Qualifier
```

QUALIFIER is used to return a code which indicates the control key used during the most recent key-press. Below is a table of of possible codes:

Table 11.8 : Qualifier codes

```
Control key   Code
===================
[None]        $8000
[Ctrl]        $8008
[Caps Lock]   $8004
Left [Shift]  $8001
Right [Shift] $8002
Left [Alt]    $8010
```

```
Right [Alt]   $8020
Left [Amiga]  $8040
Right [Amiga] $8080
```

For example:

```
; *** Qualifier example ** Filename - Qualifier.bb2

Screen 0,3
Window 0,0,0,320,200,$1000,"Amiga mode",0,1
Repeat
  ev.l=WaitEvent
  WLocate 0,0
  A$=Inkey$
  NPrint RawKey
Until A$<>""
NPrint Qualifier
VWait 50
End
```

If more than one control keys are pressed, then several codes will be returned. To read individual codes, use the logical AND operator.

11.6.2 The input cursor

These commands are used in conjunction with the EDIT and EDIT$ functions when obtaining window text input from the user.

EDITAT

```
Mode(s):  Amiga
Function: return the horizontal character position of the cursor Syntax:
x=Editat
```

This function returns the horizontal character position of the cursor. For example:

```
; *** Editat example ** Filename - Editat.bb2

Screen 0,3
Window 0,0,20,320,100,$1000,"A window",1,2
NPrint "Enter some text:"
A$=Edit$(20)
NPrint Editat
MouseWait
End
```

EDITFROM

```
Mode(s):   Amiga
Statement: control how Edit & Edit$ operate within windows
Syntax:    EditFrom [POSITION]
```

EDITFROM controls how EDIT and EDIT$ operate within windows. If the optional POSITION parameter is included then editing will begin at this character position. Example:

```
; *** EditFrom example
; *** Filename - EditFrom.bb2

Screen 0,3
Window 0,0,20,320,100,$1000,"A window",1,2
NPrint "Enter some text:"
EditFrom 16
A$=Edit$(30)
MouseWait
End
```

EDITEXIT

```
Mode(s):  Amiga/Blitz
Function: return the ASCII value of the Edit($) exit character Syntax:
e=EditExit
```

This function returns the Ascii code of the character that was used to exit a window-based EDIT or EDIT$ function. EDITFROM must be used prior to EDITEXIT to initialise alternate termination keys. Here's an example:

```
; *** EditExit example
; *** Filename - EditExit.bb2

Screen 0,3
Window 0,0,20,320,100,$1000,"",1,2
NPrint "Enter text or press escape:"
EditFrom 1
A$=Edit$(30)
If EditExit=27
  NPrint "Escape key pressed"
EndIf
MouseWait
End
```

11.7 The mouse pointer

The mouse is often used for controlling applications, especially those involving aspects of Intuition, such as windows. Here's how.

11.7.1 Mouse functions

WMOUSEX

```
Mode(s):  Amiga
Function: return horizontal mouse position relative to left of window
Syntax:   x=WMouseX
```

This function returns the horizontal position of the mouse pointer relative to the left of the currently used window. Example:

```
; *** WMouseX example
; *** Filename - WMouseX.bb2

Screen 0,2
Window 0,0,0,150,100,0,"Mouse coordinates",0,1
While Joyb(0)=0
  WLocate 0,0
  NPrint "X = ",WMouseX,"  "
Wend
End
```

To return the vertical position of the mouse pointer the corresponding WMOUSEY statement should be used.

WMOUSEY

```
Mode(s):  Amiga
Function: return vertical mouse position relative to the window top
Syntax:   y=WMouseY
```

For example:

```
; *** Mouse coordinates
; *** Filename - WMouseY.bb2

Screen 0,2
Window 0,0,0,150,100,0,"Mouse coordinates",0,1
While Joyb(0)=0
  WLocate 0,0
  NPrint "X = ",WMouseX,"  "
  NPrint "Y = ",WMouseY,"  "
Wend
End
```

EMOUSEX

```
Mode(s):  Amiga/Blitz
Function: return horizontal mouse position when last window event occured
Syntax:   x=EMouseX
```

The EMOUSEX function returns the horizontal position of the mouse pointer when the most recent "window event" occured. Window events are those which occur when a window's properties are altered. They are detected using the EVENT or WAITEVENT statements (see next example).

EMOUSEY

```
Mode(s):  Amiga/Blitz
Function: return vertical mouse position when last window event occured
Syntax:   y=EMouseY
```

The EMOUSEY function returns the vertical position of the mouse pointer when the most recent "window event" occured. Window events are those which occur when a window's properties are altered. They are detected using the EVENT or WAITEVENT statements. For example:

```
; *** Mouse coordinates 2
; *** Filename - EMouseY.bb2

Screen 0,2
ScreensBitMap 0,0
Window 0,50,50,170,100,$0008,"Close me",0,1
Repeat
  ev.l=WaitEvent
Until ev=$200
NPrint "X was ",EMouseX,"  "
NPrint "Y was ",EMouseY,"  "
VWait 100
End
```

11.7.2 Mouse buttons

MBUTTONS

```
Mode(s):  Amiga
Function: return code of the button that caused the last mouse event
Syntax:   m=MButtons
```

This function returns the code of the mouse button that caused the last "mouse button event". If menus have been turned off using the MENUS OFF statement then the right mouse button will also register an event:

Table 11.9 : MButtons return values

```
Button  Down  Up
================
Left    1     5
Right   2     6
```

Try the following example:

```
; *** MButtons example
; *** Filename - MButtons.bb2

Screen 0,3
Window 0,0,0,320,200,$1000,"Click left mouse",0,1
Repeat
  WaitEvent
Until MButtons=1
End
```

11.7.3 The mouse pointer

I've already expressed my dislike of the WIMP (Windows, Icons, Menus and Pointers) environment. So, if you feel the same way, then you'll be glad to know that the shape of the window pointer can be altered with WPOINTER.

WPOINTER

```
Mode(s):   Amiga
Statement: change the mouse pointer in the current window
Syntax:    WPointer SHAPE#
```

WPOINTER is similar to the POINTER statement in that it is used to change the shape of the mouse pointer. SHAPE# must be a previously-initialised four colour shape object:

```
; *** Point me in the right direction
; *** Filename - WPointer.bb2

Screen 0,2
Window 0,0,0,320,100,$1000,"Pointer",0,1
LoadShape 0,"POINTER.SHAPE"
WPointer 0
MouseWait
End
```

11.8 Window graphics

Blitz Basic can generate fabulous low-resolution and high-resolution window displays using its powerful drawing commands. These graphic displays are made up of small blocks of colour called pixels, and all screens are composed of thousands of pixels in varying arrangements.

WCLS

```
Mode(s):   Amiga
Statement: clear current window
Syntax:    WCls [COLOUR]
```

WCLS clears the current window with colour (0). If the optional COLOUR parameter is included then the window will be cleared with this colour. If the current window was unopened then this statement will clear the window border and title bar. For example:

```
; *** When I'm cleaning windows
; *** Filename - WCls.bb2

Screen 0,3
Window 0,0,20,320,200,$400,"A window",0,1
WBox 100,70,200,150,4
VWait 100
WCls
VWait 100
For A=1 To 50
  WCls Rnd(8)
  VWait 6
Next A
End
```

INNERCLS

```
Mode(s):   Amiga
Statement: clear inner portion of current window
Syntax:    InnerCls [COLOUR]
```

The INNERCLS statement is identical to WCLS, however it will only clear the inner portion of the current window, not the window border or title bar:

```
; *** When I'm cleaning windows take 2
; *** Filename - InnerCls.bb2

Screen 0,3
Window 0,0,20,320,200,0,"Window",0,1
VWait 100
WCls 2
VWait 100
Free Window 0
; *** Same example using InnerCls
Window 0,0,20,320,200,0,"Window",0,1
VWait 100
InnerCls 2
MouseWait
End
```

WPLOT

```
Mode(s):   Amiga
Statement: plot a single point
Syntax:    WPlot X,Y,COLOUR
```

The WPLOT statement plots a single pixel at coordintes X,Y in colour COLOUR in the currently used window. You can really only make very simple pictures with WPLOT. For example:

```
; *** WPlot example
; *** Filename - WPlot.bb2

Screen 0,3
Window 0,0,10,300,100,$0001|$1000,"",1,0
For A=1 To 100
  WPlot Rnd(270)+20,Rnd(80)+10,Rnd(6)+1
Next A
MouseWait
End
```

WLINE

```
Mode(s):   Amiga
Statement: draw a line or series of lines
Syntax:    WLine X1,Y1,X2,Y2[,XN,YN...],COLOUR
```

The WLINE statement draws a line connecting two pixels on the currently used window. X1 and Y1 specify the start of the line and X2 and Y2 specify the end of the line. If further X and Y parameters are included then polygons can be constructed. Here is an example:

```
; *** WLine examples ** Filename - WLine.bb2

Screen 0,3
Window 0,0,20,320,200,0,"",0,1
; *** Building a square with WLine
Wline 10,10,20,10,20,20,10,20,10,10,4
VWait 100
; *** Move the mouse to make pretty patterns
Repeat
  Wline SMouseX,SMouseY,Rnd(280)+20,Rnd(180)+10,Rnd(8)
Until Joyb(0)>0
MouseWait
End
```

WBOX

```
Mode(s):   Amiga
Statement: draw a solid rectangle
Syntax:    WBox X1,Y1,X2,Y2,COLOUR

WBOX is used to draw solid rectangles in the currently used window. X1 and
Y1 are the coordinates of the top left-hand corner of the rectangle and X2
and Y2 are the coordinates of the bottom right-hand corner. COLOUR
specifies the colour of the rectangle. Here is an example:

; *** WBox example
; *** Filename - WBox.bb2

Screen 0,3
Window 0,0,20,320,200,0,"Boxing clever",0,1
For A=1 To 500
  WBox Rnd(300)+10,Rnd(180)+15,Rnd(300)+5,Rnd(180)+15,Rnd(8)
Next A
MouseWait
End
```

WCIRCLE

```
Mode(s):   Amiga
Statement: draw a circular outline
Syntax:    WCircle X,Y,RADIUS,COLOUR
```

The WCIRCLE statement allows you to draw a circle in the currently used window. The position and size of the circle is set using the X and Y parameters, followed by the radius of the circle. COLOUR specifies the colour of the circle. For example:

```
; *** WCircle example
; *** Filename - WCircle.bb2

Screen 0,3
Window 0,0,20,320,200,0,"Silly circles",0,1
For A=1 To 50
  WCircle Rnd(260)+30,Rnd(100)+60,Rnd(20)+5,Rnd(8)
Next A
MouseWait
End
```

WELLIPSE

```
Mode(s):   Amiga
Statement: draw an elliptical outline
Syntax:    WEllipse X,Y,X_RADIUS,Y_RADIUS,COLOUR
```

Ellipses can be drawn just as easily with the WELLIPSE statement. WELLIPSE works in the same way as WCIRCLE, however an extra Y_RADIUS parameter is included which specifies the vertical radius of the ellipse. For example:

```
; *** WEllipse example
; *** Filename - WEllipse.bb2

Screen 0,3
Window 0,0,20,320,200,0,"Elliptical outlines",0,1
For A=1 To 90
  WEllipse Rnd(260)+30,Rnd(100)+60,Rnd(20)+5,Rnd(10)+5,Rnd(8)
Next A
MouseWait
End
```

WBLIT

```
Mode(s):   Amiga
Statement: draw a shape object in a window
Syntax:    WBlit SHAPE#,X,Y
```

This statement is used to draw, or blit, a shape object in a window, at co-ordinates (X,Y). Here's an example:

```
; *** WBlit example ** Filename - WBlit.bb2

BitMap 0,320,256,3
Boxf 10,10,20,20,5
GetaShape 0,10,10,20,20
Screen 0,3
Window 0,0,20,320,200,0,"",1,2
For A=1 To 50
  WBlit 0,Rnd(260)+30,Rnd(150)+30
Next A
Repeat
  ev.l=WaitEvent
Until ev>0
End
```

11.9 End-of-Chapter summary

A window is an independent rectangular area of text and graphics on the screen, which can accept or display information. Windows are opened using the WINDOW statement.

Windows can also be closed, moved, scrolled, cleared and re-sized.

IDCMP flags are special flags which are attached to windows. They describe the type of "event" which can be reported by a window. Events are reported by the WAITEVENT and EVENT functions.

The style and rendering mode for window text can be altered, as can the shape of the window pointer.

The window library also supports some powerful 2D drawing commands:

Table 11.10 : Window drawing commands

```
Shape      Command
===================
Square     WBOX
Rectangle  WBOX
Circle     WCIRCLE
Ellipse    WELLIPSE
```

Chapter 12 : Menus

For anyone who wishes to use their Amiga for other purposes than playing games there is a wealth of literature aimed at teaching basic programming techniques. Many who have mastered aspects of the BASIC language find themselves directed towards writing games programs rather than more serious applications. The main reason for that is the lack of direction in the literature towards developing business or educational type programs.

We can, however develop a technique for writing non-games type programs which is both simple in concept and in widespread use on all computer systems. The resulting programs come under the general category of "Menu driven programs".

Menus are lists of items, or options. Menu titles can be seen by pressing the right-hand mouse button. To observe the contents of a menu, point at the appropriate title with the mouse pointer and the options will appear directly beneath the menu title. These options are selected by moving the mouse pointer over the desired option and releasing the mouse button whilst it is highlighted.

These menus, or MenuList objects, can contain menu titles, menu items and possible even sub-menu items.

Menus are attached to windows after the window has been opened, with the SETMENU statement. Each window can have its own menus, allowing complex user interfaces to be created.

This chapter will take a look at creating menu driven programs and manipulating menus in Blitz Basic 2.

12.1 Defining menus

Each list of menu options must have a menu title, which appears at the very top of the menu.

MENUTITLE

```
Mode(s):   Amiga/Blitz
Statement: add a menu title to a MenuList
Syntax:    MenuTitle MENULIST,MENU#,TITLE$
```

This statement is used to create the Intuition menu titles which appear when the right mouse button is held down. The MENU# parameter specifies the number of the title. The title at the left-hand edge of the menu title is given a value of (0), followed by (1) for the next title and so on. TITLE$ is the text that will appear when the right mouse button is used. Here is an example:

```
; *** MenuTitle example
; *** Filename - MenuTitle.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
; *** Define menu option (see later)
MenuItem 0,0,0,0,"QUIT"
; *** Open an Intuition display
```

```
Screen 0,3
Window 0,0,20,160,100,$100f,"Blitz Basic",1,2
; *** Attach menu to window (see later)
SetMenu 0
; *** Repeat until QUIT option is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=0
End
```

12.1.1 Text menu items

Menu items are the options which appear directly beneath a menu title.

MENUITEM

```
Mode(s):   Amiga/Blitz
Statement: create a text menu item
Syntax:    MenuItem MENULIST,FLAGS,MENU#,ITEM,TEXT$[,SHORT$]
```

The MENUITEM statement creates a text menu item. Menu items are the options which appear directly below menu titles in Intuition menus. FLAGS is a special parameter which controls the status of an individual menu item:

Table 12.1 : The FLAGS parameter

```
FLAG  Description
=====================================================
0     Standard menu item
1     Toggle-type menu item (toggled by user)
2     Toggle-type menu item (toggled by FLAG 2 items)
3     As FLAG 1 but initially "On"
4     As FLAG 2 but initially "On"
```

The MENU# parameter is the number of the menu title under which the menu item will appear.

ITEM is the option number for the menu item. Menu items with an option number of (0) appear at the top of an options list, followed by (1), and so on.

TEXT$ is the option text. The optional SHORT$ parameter specifies a one character "keyboard shortcut" for the menu item. For example:

```
; *** MenuItem example
; *** Filename - MenuItem.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
; *** Define menu options
```

```
MenuItem 0,0,0,0,"Load"
MenuItem 0,0,0,1,"Save"
MenuItem 0,0,0,2,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,160,100,$100f,"Blitz Basic",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT option is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=2
End
```

```
; *** MenuItem example 2
; *** Filename - MenuItem2.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
; *** This option is a toggle item
; *** and has the keyboard short-cut
; *** right Amiga + T
MenuItem 0,3,0,0," Toggle    ","T"
MenuItem 0,0,0,1,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT option is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=1
End
```

SUBITEM

```
Mode(s):   Amiga/Blitz
Statement: create a sub-menu item
Syntax:    SubItem MENULIST,FLAGS,MENU#,ITEM,SUB,TEXT$[,SHORT$]
```

All menu items may have an optional list of sub-menu items attached to them. This is where the SUBITEM statement comes in.

The ITEM parameter specifies the menu item to attach the sub item to. SUB is the index number for this sub item, those with number (0) appear at the top of the sub item list, followed by (1) and so on. The TEXT$ parameter is the sub item text.

As with menu items, sub items may have keyboard shortcuts attached with the optional SHORT$ parameter (see the MENUITEM statement for details of the other parameters):

```
; *** SubItem example
; *** Filename - SubItem.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
; *** Define menu option 0
MenuItem 0,0,0,0,"Load"
; *** Define menu option 0 sub-options
SubItem 0,0,0,0,0,"Picture"
SubItem 0,0,0,0,1,"Sample"
; *** Define menu option 1
MenuItem 0,0,0,1,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT option is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=1
End
```

12.1.2 Shape menu items

Shape menu items are the graphical elements which appear directly beneath menu titles. They use pictures, instead of words, as options.

SHAPEITEM

```
Mode(s):   Amiga/Blitz
Statement: create a graphical menu item
Syntax:    ShapeItem MENULIST,FLAGS,MENU#,ITEM,SHAPE#
```

The SHAPEITEM statement is used to create a graphical menu item. As with the MENUITEM statement, the FLAGS parameter controls the status of an individual menu item:

Table 12.2 : The FLAGS parameter

```
FLAG  Description
=====================================================
0     Standard menu item
1     Toggle-type menu item (toggled by user)
2     Toggle-type menu item (toggled by FLAG 2 items)
3     As FLAG 1 but initially "On"
4     As FLAG 2 but initially "On"
```

The MENU# parameter is the number of the menu title under which the menu item will appear. ITEM is the option number for the menu item. Menu items with an option number of (0) appear at the top of an options list, followed by (1), and so on.

SHAPE# specifies the number of a previously initialised shape object to be used as the graphics. Here is an example:

```
; *** ShapeItem example
; *** Filename - ShapeItem.bb2

; *** Open screen and grab its BitMap
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
; *** Create a shape
Cls 2
Boxf 30,30,60,60,5
Circlef 100,40,10,6
GetaShape 0,30,30,60,60
GetaShape 1,90,30,110,50
Cls
; *** Define menu title
MenuTitle 0,0,"Shape item"
; *** Define shape options
ShapeItem 0,0,0,0,0
ShapeItem 0,0,0,1,1
; *** Open Intuition window
Window 0,0,20,200,100,$100f,"Select a menu",1,2
; *** Attach menu to window
SetMenu 0
; *** Circle item quits
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=1
End
```

SHAPESUB

```
Mode(s):   Amiga/Blitz
Statement: create a graphic sub-menu item
Syntax:    ShapeSub MENULIST,FLAGS,MENU#,ITEM,SUBITEM,SHAPE#
```

SHAPESUB creates a graphical sub-menu item. The ITEM parameter specifies the menu item to attach the sub item to. SUBITEM is the index number for this sub item, those with number (0) appear at the top of the sub item list, followed by (1) and so on. The SUBTEXT$ parameter is the sub item text.

As with menu items, sub items may have keyboard shortcuts attached with the optional SHORTCUT$ parameter (see the SHAPEITEM statement for details of the other parameters). SHAPE# is the number of a previously initialised shape object to be used as the graphic. For example:

```
; *** ShapeSub example
; *** Filename - ShapeSub.bb2

; *** Open screen and grab its BitMap
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
; *** Create a shape
Cls 2
Boxf 30,30,60,60,5
Circlef 100,40,10,6
GetaShape 0,30,30,60,60
GetaShape 1,90,30,110,50
Cls
; *** Define menu title
MenuTitle 0,0,"Shape item"
; *** Define shape option
ShapeItem 0,0,0,0,0
; *** Define shape sub-option
ShapeSub 0,0,0,0,0,1
; *** Open Intuition window
Window 0,0,20,200,100,$100f,"Select a menu",1,2
; *** Attach menu to window
SetMenu 0
; *** Circle item quits
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=0
End
```

12.2 Creating menus

To make a menu visible it must be attached to a window with the SETMENU statement.

SETMENU

```
Mode(s):   Amiga
Statement: attach a MenuList to the current window
Syntax:    SetMenu MENULIST
```

This statement attaches a MenuList to the currently used window. Each window may have only one MenuList attached to it. For example:

```
; *** SetMenu example
; *** Filename - SetMenu.bb2

; *** Define a menu
MenuTitle 0,0,"Project"
```

```
MenuItem 0,0,0,0,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT option is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=0
End
```

12.3 Manipulating menus

The state, layout and colour of menus and menu options can be altered with the following MenuList commands.

MENUCOLOUR

```
Mode(s):   Amiga/Blitz
Statement: determine the colour of a menu item
Syntax:    MenuColour COLOUR
```

The MENUCOLOUR statement is used to set the colour of the text in a menu item or sub item. Here is an example:

```
; *** MenuColour example
; *** Filename - MenuColour.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
; *** Alter text colour
MenuColour 3
MenuItem 0,0,0,0,"I want to"
; *** Alter text colour again
MenuColour 5
MenuItem 0,0,0,1,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT option is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=1
End
```

MENUGAP

```
Mode(s):   Amiga/Blitz
Statement: control layout of a menu
Syntax:    MenuGap X_GAP,Y_GAP
```

Executing MENUGAP before creating any menu titles, items or sub items, allows you to control the layout of the menu. The X_GAP and Y_GAP parameters specify an amount, in pixels, to be inserted to the left and right of all menu items and sub-menu items. Try the following example:

```
; *** MenuGap example
; *** Filename - MenuGap.bb2

; *** Set menu gap
MenuGap 60,30
; *** Define menu title
MenuTitle 0,0,"Project"
MenuItem 0,0,0,0,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT optin is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=0
End
```

SUBITEMOFF

```
Mode(s):   Amiga/Blitz
Statement: control the position of sub items to menu options
Syntax:    SubItemOff X_OFFSET,Y_OFFSET
```

The SUBITEMOFF statement controls the relative position of the top of a list of sub-menu items in relation to their associated menu item:

```
; *** SubItemOff example
; *** Filename - SubItemOff.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
MenuItem 0,0,0,0,"I want to"
; *** Set option offset
SubItemOff 80,80
```

```
SubItem 0,0,0,0,0,"Quit"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
; *** Repeat until QUIT optin is selected
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=0
End
```

MENUSTATE

```
Mode(s):   Amiga/Blitz
Statement: turn menu items on or off
Syntax:    MenuState MENULIST[,MENU#[,ITEM[,SUB]]],On/Off
```

This statement is used to turn entire menus or parts of menus on or off. If the following syntax is used then a whole active menu may be turned off:

```
MenuState MENULIST On/Off
```

If the MENU# parameter is included then a menu may be turned on or off:

```
MenuState MENULIST,MENU#,On/Off
```

Menu items and sub items can also be toggled by the inclusion of the appropriate parameters. Try the following example which illustrates this:

```
; *** MenuState example
; *** Filename - MenuState.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
MenuColour 3
; ** This item will be turned off
MenuItem 0,0,0,0,"Load"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
MenuState 0,0,0,Off
; *** Wait for a mouse click
```

```
Repeat
Until WaitEvent=$8
End
```

MENUCHECKED

```
Mode(s):  Amiga/Blitz
Function: read status of a toggle-type menu item
Syntax:   s=MenuChecked(MENULIST,MENU#,ITEM[,SUBITEM])
```

This function reads the status of a toggle-type menu item or menu sub item. If the item is currently "checked" then MENUCHECKED wil return (-1), otherwise (0) will be returned. For example:

```
; *** MenuChecked example
; *** Filename - MenuChecked.bb2

; *** Define menu title
MenuTitle 0,0,"Project"
MenuColour 3
MenuItem 0,1,0,0," Enable quit"
MenuItem 0,0,0,1,"QUIT"
; *** Open an Intuition display
Screen 0,3
Window 0,0,20,200,100,$100f,"Press right mouse",1,2
; *** Attach menu to window
SetMenu 0
Repeat
  ev.l=WaitEvent
  ; *** End if QUIT is enabled
  If ev=256 AND ItemHit=1
    If MenuChecked(0,0,0)
      End
    ; *** QUIT option is not enabled
    Else
      WLocate 0,0
      Print "Quit not enabled"
    EndIf
  EndIf
Forever
```

12.4 A full example

Here is a full example which demonstrates one of the practical applications of menus in Blitz Basic programs. It is a very simple IFF image displayer. At present it can only load IFF files and clear them from the screen. Why not try adding the ability to save screens, or add more options, such as screen distortion and palette manipulation:

```
; *** Menu example
; *** Filename - MoreMenus.bb2

; *** First menu
MenuTitle 0,0,"PROJECT"
MenuItem 0,0,0,0,"LOAD    ","L"
MenuItem 0,0,0,1,"SAVE    ","S"
MenuItem 0,0,0,2,"QUIT    ","Q"
; *** Second menu
MenuTitle 0,1,"SPECIAL"
MenuItem 0,0,1,0,"CLEAR    ","C"
; *** Output screen and window
Screen 0,0,0,320,200,5,0,"Complete menu example",1,7
Window 0,0,0,320,200,$1900,"",1,0
; *** Attach MenuList to window
SetMenu 0
Repeat
  A.l=WaitEvent
  If A=256
    ; *** "LOAD" option
    If MenuHit=0 AND ItemHit=0
      MaxLen PATH$=160
      MaxLen NAME$=64
      A$=FileRequest$("Load",PATH$,NAME$)
      If A$<>""
        LoadScreen 0,A$,0
        Use Palette 0
      EndIf
    EndIf
    ; *** "QUIT" option
    If MenuHit=0 AND ItemHit=2 Then End
    ; *** "CLEAR" option
    If MenuHit=1 AND ItemHit=0 Then WCls
  EndIf
Forever
```

12.5 End-of-Chapter summary

Menus are created through the use of MenuList objects. Each MenuList contains an entire set of menu titles, menu items and possibly sub-menu items.

MENUTITLE is used to create the Intuition menu titles which appear when the right mouse button is held down.

The **MENUITEM** statement creates a text menu item.

The **SHAPEITEM** statement is used to create a graphical menu item.

All menu items may have an optional list of sub-menu items attached to them. This is achieved with the SUBITEM and SHAPESUBITEM statements.

Menus are attached to windows by the SETMENU statement.

The layout of menus can be altered with MENUGAP and SUBITEMOFF.

Menu status can be read and altered with the MENUSTATE and MENUCHECKED commands.

Table 12.3 : Menu commands

Command	Function
MENUCHECKED	Read status of a toggle-type menu item
MENUCOLOUR	Determine the colour of a menu item
MENUGAP	Control layout of a menu
MENUITEM	Create a text menu item
MENUSTATE	Turn menu items on or off
MENUTITLE	Add a menu title to a MenuList
SETMENU	Attach a MenuList to the current window
SHAPEITEM	Create a graphical menu item
SHAPESUB	Create a graphic sub-menu item
SUBITEM	Create a sub-menu item
SUBITEMOFF	Control position of sub items to menu options

Chapter 13 : Gadgets

Gadgets are the boxes which appear when the program requires you to enter or alter information. They are selected by clicking on the gadget once with the mouse pointer, although some gadgets require you to enter text (string gadgets).

Note that all Blitz gadgets are created inside a previously initialised window. As a brief reminder, the syntax of the WINDOW statement is as follows:

```
Window WINDOW#,X,Y,W,H,F,TITLE$,D,B[,G_LIST#[,BITMAP#]]
```

Gadgets are created through the use of GadgetList objects. This allows you to group several gadgets, with differing identification numbers, under one GadgetList. This requires the presence of yet another parameter in the window's definition.

The optional G_LIST# parameter is the number of a GadgetList object to be attached to the window. For example:

```
; *** Gadgets example
; *** Filename - Gadgets.bb2

TextGadget 0,60,100,0,1," Quit "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

All Blitz Basic gadgets need X and Y parameters, and some require you to specify the size of the gadget, in pixels.

Let's begin with text gadgets, the simplest in Blitz Basic's repertoire...

13.1 Text gadgets

TEXTGADGET

```
Mode(s):   Amiga/Blitz
Statement: add a text gadget to a GadgetList
Syntax:    TextGadget GADGETLIST,X,Y,FLAGS,ID,TEXT$
```

The TEXTGADGET statement adds a text gadget to a GadgetList. A text gadget is the simplest type of gadget, consisting of a sequence of characters surrounded by an optional border. X and Y specify the co-ordinates of the gadget in the currently used window. The FLAGS parameter should be set as follows:

Table 13.1 : The FLAGS parameter

```
Bit#  Description
==================================
0     Toggle on/off
1     Relative to right of window
2     Relative to bottom of window
5     Box select
```

If BIT# 1 is set then the X parameter should be negative, and if BIT# 2 is set then the Y parameter should be negative.

ID is a specific identification number, greater than zero, attached to this gadget. The TEXT$ parameter holds the actual text string to appear in the gadget. Here is a full example:

```
; *** TextGadget example
; *** Filename - TextGadget.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,170,100,0,2," Don't quit "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

GADGETPENS

```
Mode(s):   Amiga/Blitz
Statement: define text colours for text gadgets
Syntax:    GadgetPens FOREGROUND[,BACKGROUND]
```

This statement sets the text colours used when text gadgets are created using the TEXTGADGET statement. Obviously, the FOREGROUND parameter defines the foreground colour (default colour is 1) and the optional BACKGROUND parameter, the background colour (default colour is 0). Try the following example:

```
; *** GadgetPens example
; *** Filename - GadgetPens.bb2

GadgetPens 6
TextGadget 0,60,100,0,1," Quit "
GadgetPens 4,5
```

```
TextGadget 0,170,100,0,2," Don't quit "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

GADGETJAM

```
Mode(s):   Amiga/Blitz
Statement: determine text rendering method for text gadgets
Syntax:    GadgetJam MODE
```

GADGETJAM controls the text rendering method when creating text gadgets. The MODE parameter is identical to that of the WJAM statement in Chapter 11. Here is a brief reminder:

Table 13.2 : The MODE parameter

```
Value  Mode        Description
========================================================================
0      Jam1        Only the foreground colour to be printed
1      Jam2        Print foreground and background colours
2      Complement  Print old characters that overlap with new ones (XOR)
4      Inverse     Print inverse video characters
```

Try the following example:

```
; *** GadgetJam example
; *** Filename - GadgetJam.bb2

TextGadget 0,60,100,0,1," Quit "
GadgetJam 4
TextGadget 0,150,100,0,2," Inverse video "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

13.1.1 Cycling text gadgets

Cycling gadgets are similar to text gadgets, however when they are selected by the user, another text string is displayed in the gadget.

To create a cycling text gadget, use the "|" character in the TEXT$ parameter to seperate the options:

```
; *** TextGadget example 2
; *** Filename - TextGadget2.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,170,100,0,2,"CYCLE |GADGET"
Screen 0,3
Window 0,0,20,320,200,$100f,"Cycle the gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

SETGADGETSTATUS

```
Mode(s):   Amiga
Statement: set cycling gadget
Syntax:    SetGadgetStatus GADGETLIST#,ID,VALUE
```

This statement sets the status of a cycling text gadget. The VALUE parameter is the number of the text item to be displayed. The REDRAW statement should be used to update the gadget. Here's an example:

```
; *** SetGadgetStatus example
; *** Filename - SetGadgetStatus.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,140,100,0,2,"First Option|Second option"
Screen 0,3
Window 0,0,20,320,200,$100f,"Window",1,2,0
SetGadgetStatus 0,2,2
Redraw 0,2
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

GADGETSTATUS

```
Mode(s):  Amiga
Function: return gadget status
Syntax:   GadgetStatus(GADGETLIST#,ID)
```

This function returns the status of a gadget. If the specified gadget is a "toggle" type gadget then GADGETSTATUS will return (-1) if the gadget is on, or (0) if the gadget is off. If the specified gadget is a "cycling text" type gadget, then GADGETSTATUS will return a value representing the text item number. For example:

```
; *** GadgetStatus example
; *** Filename - GadgetStatus.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,170,100,0,2,"Blitz|Basic| Two "
Screen 0,3
Window 0,0,20,320,200,$100f,"Cycle Blitz gadget",1,2,0
Repeat
  WLocate 0,0
  NPrint "Cycle text ",GadgetStatus(0,2)
Until WaitEvent=64 AND GadgetHit=1
End
```

13.2 Shape gadgets

The SHAPEGADGET statement is used to create gadgets with graphic elements, taken from a previously initialised shape bank.

SHAPEGADGET

```
Mode(s):   Amiga/Blitz
Statement: add a shape gadget to a GadgetList
Syntax:    ShapeGadget GADGETLIST,X,Y,FLAGS,ID,SHAPE#[,SHAPE2#]
```

SHAPE# is the number of the shape object to appear in the gadget. The FLAGS parameter should be set as follows:

Table 13.3 : The FLAGS parameter

```
Bit#  Description
==================================
0     Toggle on/off
1     Relative to right of window
2     Relative to bottom of window
5     Box select
```

All of the other parameters are the same as for the TEXTGADGET statement. If the optional SHAPE2# parameter is included then an alternative shape may be displayed when the gadget is selected. For example:

```
; *** ShapeGadget example
; *** Filename - ShapeGadget.bb2

Screen 0,3
ScreensBitMap 0,0
```

```
For A=7 To 1 Step -1
  Circlef 16,32,A*2,A
Next
GetaShape 0,0,16,32,32
Cls
ShapeGadget 0,148,50,0,1,0
TextGadget 0,140,180,0,2," Quit "
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=2
End
```

TOGGLE

```
Mode(s):   Amiga/Blitz
Statement: turn a text or shape object on or off
Syntax:    Toggle GADGETLIST,ID[,On/Off]
```

This statement is used to turn a toggle-type shape or text gadget on or off. If the optional On/Off parameter is missing then the gadget will be toggled to its alternate state (i.e. if the gadget is currently on then it will be toggled off, and vice versa). Toggle-type gadgets are those created with BIT# 0 set (consult the TEXTGADGET statement for more information):

```
; *** Toggle example
; *** Filename - Toggle.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,170,100,0,2," Toggled On "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
Toggle 0,2,On
Redraw 0,2
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

13.3 String gadgets

The STRINGGADGET statement is used to create an Intuition "text input" gadget. When a string gadget is selected, a text cursor appears and characters may be input into the gadget from the keyboard.

STRINGGADGET

```
Mode(s):   Amiga/Blitz
Statement: add a text entry gadget to a GadgetList
Syntax:    StringGadget GADGETLIST,X,Y,FLAGS,ID,LENGTH,WIDTH
```

X and Y control the gadgets horizontal and vertical position, in pixels, relative to the top left of the currently used window.

FLAGS should be set as follows:

Table 13.4 : The FLAGS parameter

```
Bit#  Description
==================================
1     Relative to right of window
2     Relative to bottom of window
5     Box select
```

If BIT# 1 is set then the X parameter should be negative, and if BIT# 2 is set then the Y parameter should be negative.

The ID parameter is the number of the gadget.

The LENGTH parameter specifies the maximum number of characters that may appear in the gadget.

WIDTH specifies the width of the string gadget in pixels. If WIDTH is less than LENGTH then excess characters will be scrolled in the gadget.

Here is a full example:

```
; *** StringGadget example
; *** Filename - StringGadget.bb2

StringGadget 0,80,16,0,1,40,160
StringGadget 0,80,32,0,2,40,160
TextGadget 0,8,180,0,3," QUIT "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
WLocate 8,8
Print "Name:"
WLocate 8,24
Print "Address:"
Repeat
Until WaitEvent=64 AND GadgetHit=3
End
```

13.3.1 Manipulating string gadgets

STRINGTEXT$

```
Mode(s):  Amiga/Blitz
Function: return the contents of a string gadget
Syntax:   c=StringText$(GADGETLIST,ID)
```

This function returns the contents of a string gadget. The GADGETLIST parameter specifies the number of the GadgetList and the ID parameter specifies the number of the string gadget. For example:

```
; *** StringText$ example
; *** Filename - StringText$.bb2

StringGadget 0,80,16,0,1,40,160
TextGadget 0,8,180,0,2," QUIT "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
WLocate 4,8
Print "Name:"
Repeat
  ev.l=WaitEvent
  If ev=64 AND GadgetHit=1
    WLocate 8,96
    Print "Hello there "+StringText$(0,1)
    ClearString 0,1
    Redraw 0,1
  EndIf
Until ev=64 AND GadgetHit=2
End
```

ACTIVATESTRING

```
Mode(s):   Amiga/Blitz
Statement: activate a string gadget
Syntax:    ActivateString WINDOW#,ID
```

The ACTIVATESTRING statement is used to automatically activate a string gadget. Here is an example:

```
; *** ActivateString example
; *** Filename - ActivateString.bb2

StringGadget 0,80,16,0,1,40,160
TextGadget 0,8,180,0,2," QUIT "
Screen 0,3
```

```
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
WLocate 4,8
Print "Type:"
ActivateString 0,1
Repeat
  ev.l=WaitEvent
Until ev=64 AND GadgetHit=2
End
```

RESETSTRING

```
Mode(s):   Amiga/Blitz
Statement: reset a string gadget
Syntax:    ResetString GADGETLIST,ID
```

RESETSTRING resets a string gadget and moves its cursor position to the left of the gadget.

CLEARSTRING

```
Mode(s):   Amiga/Blitz
Statement: erase the text in a string gadget
Syntax:    ClearString GADGETLIST,ID
```

This statement erases the contents of a string gadget (ID). If a string gadget is erased while it is displayed in a window then the text will not be erased until the REDRAW statement is executed.

SETSTRING

```
Mode(s):   Amiga/Blitz
Statement: initialize a string gadget
Syntax:    SetString GADGETLIST,ID,STRING$
```

SETSTRING is used to initialise the contents of a string gadget created using the STRINGGADGET statement. REDRAW may be used to update any changes.

Try the following example which demonstrates RESETSTRING, CLEARSTRING and SETSTRING:

```
; *** Manipulating string gadgets
; *** Filename - SetString.bb2

StringGadget 0,80,16,0,1,40,160
TextGadget 0,8,180,0,2," QUIT "
Screen 0,3
Window 0,0,20,320,200,$100f,"Select a gadget",1,2,0
WLocate 4,8
```

```
Print "Name:"
ActivateString 0,1
Repeat
  ev.l=WaitEvent
  If ev=64 AND GadgetHit=1
    ResetString 0,1
    ClearString 0,1
    SetString 0,1,"is cool!"
    ActivateString 0,1
  EndIf
Until ev=64 AND GadgetHit=2
End
```

13.4 Gadget groups

If a gadget group is defined then all button gadgets created become part of a mutually exclusive group. This means that, if one gadget is selected, then all other gadgets of the same group become unselected.

BUTTONGROUP

```
Mode(s):   Amiga
Statement: define a number of button-type gadgets
Syntax:    ButtonGroup GROUP
```

This statement is used to define a number of gadgets as being of the same, mutually exclusive group. After BUTTONGROUP has been executed, all button gadgets created become part of this group. Try the following example:

```
; *** ButtonGroup example
; *** Filename - ButtonGroup.bb2

Screen 0,10
TextGadget 0,48,14,0,0,"Cycle 1|Cycle 2|Cycle 3"
ButtonGroup 1
For A=1 To 5
  TextGadget 0,48,14+A*14,512,A,"Button "+Str$(A)
Next A
Window 0,20,20,160,100,$1008,"ButtonGroup",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

BUTTONID

```
Mode(s):  Amiga
Function: return selected gadget in button group
Syntax:   ButtonId(GADGETLIST#,BUTTONGROUP)
```

This function returns which gadget in a button group is currently selected. Here's an example:

```
; *** ButtonId example
; *** Filename - ButtonId.bb2

Screen 0,10
ButtonGroup 1
For A=1 To 5
  TextGadget 0,48,14+A*14,512,A,"Button "+Str$(A)
Next A
Window 0,20,20,320,100,$1008,"ButtonId",1,2,0
Repeat
  WLocate 45,4
  NPrint "Button ",ButtonId(0,1)," "
  ev.l=WaitEvent
Until ev=$200
End
```

13.5 Proportional gadgets

Proportional gadgets are commonly known as "slider bars", such as the gadgets that are used to scroll the contents of Workbench windows around.

Prop gadgets have two main qualities: a potentiometer setting (or pot), and a body setting. The pot setting specifies the current position of the slider bar, in the range zero to one. For example, a proportional gadget with a pot setting of 0.5 would be positioned centrally. The body setting specifies the size of the units the proportional gadget represents, in the range zero to one. The body setting also alters the width of the slider bar.

Proportional gadgets may be either horizontal or vertical, or a combination of both.

PROPGADGET

```
Mode(s):   Amiga/Blitz
Statement: create a proportional gadget
Syntax:    PropGadget GADGETLIST,X,Y,FLAGS,ID,WIDTH,HEIGHT
```

This statement is used to create a proportional gadget. The X and Y parameters specify the position of the gadget, relative to the top left of the currently used window. WIDTH and HEIGHT specify the size of the proportional gadget, in pixels. ID is simply the identification number of the gadget.

The FLAGS parameter should be set as follows:

Table 13.5 : The FLAGS parameter

```
Bit#  Description
====================================
1     Relative to right of window
2     Relative to bottom of window
3     Size relative to window width
4     Size relative to window height
5     Box select
6     Prop gadget has X movement
7     Prop gadget has Y movement
8     No border around prop gadget
```

If BIT# 1 is set then the X parameter should be negative, and if BIT# 2 is set then the Y parameter should be negative. Here are some examples:

```
; *** Proportional gadgets
; *** Filename - PropGadget.bb2

FindScreen 0
PropGadget 0,130,25,128,0,16,54
Window 0,0,20,300,100,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

```
; *** Proportional gadgets 2
; *** Filename - PropGadget2.bb2

Screen 0,3
PropGadget 0,100,25,8+(7*8),0,100,50
Window 0,0,20,300,200,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

SETHPROP

```
Mode(s):   Amiga/Blitz
Statement: change the horizontal slider qualities of a proportional gadget
Syntax:    SetHProp GADGETLIST,ID,POT,BODY
```

The SETHPROP statement is used to change the horizontal slider qualities of a proportional gadget. REDRAW may be used to update any changes. Here are some examples:

```
; *** SetHProp example
; *** Filename - SetHProp.bb2

FindScreen 0
PropGadget 0,130,25,128-64,0,120,24
; *** Position slider at left
SetHProp 0,0,0,0.3
Window 0,0,20,640,100,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

```
; *** SetHProp example 2
; *** Filename - SetHProp2.bb2

FindScreen 0
PropGadget 0,130,25,128-64,0,120,24
; *** Position slider at right
SetHProp 0,0,1,0.3
Window 0,0,20,640,100,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

```
; *** SetHProp example 3
; *** Filename - SetHProp3.bb2

FindScreen 0
PropGadget 0,130,25,128-64,0,120,24
; *** Change width of slider
SetHProp 0,0,0.5,0.1
Window 0,0,20,640,100,$100A,"",1,2,0
Repeat
```

```
  ev.l=WaitEvent
Until ev=$200
End
```

SETVPROP

```
Mode(s):   Amiga/Blitz
Statement: change the vertical slider qualities of a proportional gadget
Syntax:    SetVProp GADGETLIST,ID,POT,BODY
```

The SETVPROP statement is used to change the vertical slider qualities of a proportional gadget. REDRAW may be used to update any changes. Here are some examples:

```
; *** SetVProp example
; *** Filename - SetVProp.bb2

FindScreen 0
PropGadget 0,130,25,128,0,16,54
; *** Position pot at top
SetVProp 0,0,0,0.5
Window 0,0,20,640,200,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

```
; *** SetVProp example 2
; *** Filename - SetVProp2.bb2

FindScreen 0
PropGadget 0,130,25,128,0,16,54
; *** Position pot at bottom
SetVProp 0,0,1,0.5
Window 0,0,20,640,200,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

```
; *** SetVProp example 3
; *** Filename - SetVProp3.bb2

FindScreen 0
PropGadget 0,130,25,128,0,16,54
```

```
; *** Smaller width of slider bar
SetVProp 0,0,0.5,0.1
Window 0,0,20,640,200,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

HPROPPOT

```
Mode(s):  Amiga/Blitz
Function: return the horizontal pot setting of a proportional gadget
Syntax:   hp=HPropPot(GADGETLIST,ID)
```

This function returns a number ranging from zero, up to but not including one, reflecting the horizontal "pot" setting of a proportional gadget. For example:

```
; *** HPropPot example
; *** Filename - HPropPot.bb2

FindScreen 0
PropGadget 0,130,25,128-64,0,120,24
SetHProp 0,0,0.5,0.2
Window 0,0,20,640,100,$100A,"",1,2,0
Repeat
  ev.l=WaitEvent
  WLocate 0,0
  NPrint HPropPot(0,0),"    "
Until ev=$200
End
```

HPROPBODY

```
Mode(s):  Amiga/Blitz
Function: return the horizontal body setting of a proportional gadget
Syntax:   hb=HPropBody(GADGETLIST,ID)
```

This function returns a number ranging from zero, up to but not including one, reflecting the horizontal "body" setting of a proportional gadget. Example:

```
; *** HPropBody example
; *** Filename - HPropBody.bb2

FindScreen 0
PropGadget 0,130,25,128-64,0,120,24
SetHProp 0,0,0.5,0.3
Window 0,0,20,640,100,$100A,"",1,2,0
WLocate 0,0
; *** Returns 0.3
NPrint HPropBody(0,0),"    "
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

VPROPPOT

```
Mode(s):  Amiga/Blitz
Function: return the vertical pot setting of a proportional gadget
Syntax:   vp=VPropPot(GADGETLIST,ID)
```

This function returns a number ranging from zero, up to but not including one, reflecting the vertical "pot" setting of a proportional gadget. Try this example:

```
; *** VPropPot example
; *** Filename - VPropPot.bb2

FindScreen 0
PropGadget 0,130,25,128,0,16,54
Window 0,0,20,300,100,$100A,"",1,2,0
Repeat
  V=VPropPot(0,0)
  WLocate 0,0
  NPrint V,"   "
  ev.l=WaitEvent
Until ev=$200
End
```

VPROPBODY

```
Mode(s):  Amiga/Blitz
Function: return the vertical body setting of a proportional gadget
Syntax:   vb=VPropBody(GADGETLIST,ID)
```

This function returns a number ranging from zero, up to but not including one, reflecting the vertical "body" setting of a proportional gadget. For example:

```
; *** VPropBody example
; *** Filename - VPropBody.bb2

FindScreen 0
PropGadget 0,130,25,128,0,32,64
SetVProp 0,0,0.5,0.3
Window 0,0,20,640,100,$100A,"",1,2,0
WLocate 0,0
; *** Returns 0.3
NPrint VPropBody(0,0),"    "
Repeat
  ev.l=WaitEvent
Until ev=$200
End
```

Here is a full example which demonstrates the use of proportional-type gadgets:

```
; *** Palette Requester
; *** Filename - PaletteRequester.bb2

FindScreen 0
For A=0 To 2
  PropGadget 0,A*22+8,14,128,A,16,54
Next A
For B=0 To 3
  GadgetJam 1
  GadgetPens 0,B
  X=B AND 7
  Y=Int(B/8)
  TextGadget 0,X*28+72,14+Y*14,32,3+B,"  "
Next B
Window 0,100,50,300,72,$100A,"Palette requester",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
End
```

REDRAW

```
Mode(s):   Amiga/Blitz
Statement: redisplay a gadget
Syntax:    Redraw WINDOW#,ID
```

The REDRAW statement redisplays a gadget in the specified window. This is primarily of use when proportional gadgets or string gadgets have been altered. Example:

```
; *** Redraw example
; *** Filename - Redraw.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,140,100,0,2,"First Option|Second option"
Screen 0,3
Window 0,0,20,320,200,$100f,"Window",1,2,0
SetGadgetStatus 0,2,2
Redraw 0,2
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

13.6 Gadget borders

BORDERS

```
Mode(s):   Amiga/Blitz
Statement: toggle automatic border creation/specify gadget-border spacing
Syntax:    Borders On/Off
Syntax 2:  Borders WIDTH,HEIGHT
```

BORDERS is one of a number of Blitz Basic statements with two purposes. The first syntax is used to toggle automatic gadget border creation. Gadget borders are created when the TEXTGADGET or STRINGGADGET statements are used. To disable this process use:

```
Borders Off
```

The BORDERS statement may also be used to specify the spacing between a gadget and its border. In this case, the WIDTH parameter refers to the left/right spacing and HEIGHT refers to the top/bottom spacing. Try the following example:

```
; *** Flower Borders
; *** Filename - Borders.bb2

Borders Off
TextGadget 0,8,16,0,1,"No borders"
Borders On
TextGadget 0,8,32,0,2,"Borders"
Borders 16,8
TextGadget 0,8,64,0,3,"Big borders"
Borders 8,4
TextGadget 0,8,180,0,4," QUIT "
Screen 0,3
Window 0,0,20,320,200,$100f,"Sliders",1,3,0
Repeat
Until WaitEvent=64 AND GadgetHit=4
End
```

BORDERPENS

```
Mode(s):   Amiga/Blitz
Statement: define the colours used when gadget borders are created
Syntax:    BorderPens HILIGHT_COLOUR,SHADOW_COLOUR
```

The BORDERPENS statement is used to define the colours used when gadget borders are created using TEXTGADGET, STRINGGADGET and GADGETBORDER. The HILIGHT_COLOUR parameter specifies the colour of the top and left edges of the border (default colour is 1) and SHADOW_COLOUR specifies the colour of the right and bottom edges (default colour is 2). For example:

```
; *** BorderPens examples
; *** Filename - BorderPens.bb2

Borders Off
TextGadget 0,8,16,0,1,"No border"
Borders On
BorderPens 2,1
TextGadget 0,8,32,0,2,"Indented"
Borders 16,8
BorderPens 4,5
```

```
TextGadget 0,8,64,0,3,"Jolly"
Borders 8,4
BorderPens 1,2
TextGadget 0,8,180,0,4," QUIT "
Screen 0,3
Window 0,0,20,320,200,$100f,"Sliders",1,3,0
Repeat
Until WaitEvent=64 AND GadgetHit=4
End
```

GADGETBORDER

```
Mode(s):   Amiga/Blitz
Statement: draw a rectangular border into the current window
Syntax:    GadgetBorder X,Y,WIDTH,HEIGHT
```

This statement draws a rectangular border into the currently used window. Both proportional gadgets and shape gadgets do not have borders automatically created for them, so GADGETBORDER can be employed to do this. The X and Y parameters are the coordinates for the top left-hand corner of the border, and WIDTH and HEIGHT specify the width and height, in pixels, of the border. Here is a full example:

```
; *** GadgetBorder example
; *** Filename - GadgetBorder.bb2

Borders Off
TextGadget 0,8,16,0,1," No border "
Borders On
TextGadget 0,8,32,0,2," QUIT "
Screen 0,3
Window 0,0,20,320,200,$100f,"Sliders",1,3,0
VWait 100
GadgetBorder 12,15,80,10
Repeat
Until WaitEvent=64 AND GadgetHit=2
End
```

13.7 Disabling gadgets

These two statements can disable and enable gadgets respectively. If a gadget is disabled then it is covered by a "mesh" and cannot be accessed by the user. Once a gadget is enabled, this mesh is removed and the gadget can be accessed.

DISABLE

```
Mode(s):   Amiga
Statement: disable gadget
Syntax:    Disable GADGETLIST#,ID
```

ENABLE

```
Mode(s):   Amiga
Statement: enable gadget
Syntax:    Enable GADGETLIST#,ID
```

Example:

```
; *** Disable example
; *** Filename - Disable.bb2

TextGadget 0,60,100,0,1," Quit "
TextGadget 0,140,100,0,2,"Disabled gadget"
Disable 0,2
Screen 0,3
Window 0,0,20,320,200,$100f,"Window",1,2,0
Repeat
Until WaitEvent=64 AND GadgetHit=1
End
```

13.8 The GadTools Library

The GadTools library is an extension to the Blitz Basic system, which requires Kickstart 2.04 or greater. GadTools allows you to create more functional interfaces, involving list gadgets, cycle gadgets and highlight gadgets.

13.8.1 Basics of GadTools

All GadTools gadgets have a unique ID number, enabling you to mix standard Blitz gadgets with GadTools.

GadTools gadgets have X, Y, W (width) and H (height) parameters which specify the location and dimensions of the gadget. They also have a TEXT$ parameter which is used as a title for the gadget.

Unlike standard Blitz Basic gadgets, GadTools gadgets are attached to the window after it has been opened. This is achieved with the ATTACHGTLIST statement.

ATTACHGTLIST

```
Mode(s):   Amiga
Statement: attach GadTools gadgets to a window
Syntax:    AttachGTList GTLIST#,WINDOW#
```

This statement is used to attach a series of GadTools gadgets to a window, after it has been opened. Here's an example:

```
; *** AttachGTList example
; *** Filename - AttachGTList.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
Window 0,0,20,320,70,$1000,"A simple example",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

In all of the GadTools commands, where a FLAGS parameter is required, it should be set as follows:

Table 13.6 : The FLAGS parameter

Flag	Description
1	Text label to left of gadget
2	Text label to right of gadget
4	Text label above gadget
8	Text label below gadget
$10	Text label inside gadget
$20	Gadget highlighted
$40	Gadget disabled
$80	Immediate flag
$100	Set Boolean gadget to (On)
$200	Attach arrows to scroller gadget
$400	Make GTPROPGADGET vertical

Here are some examples:

```
; *** FLAGS example 1
; *** Filename - FLAGS.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTNumber 0,1,150,30,60,20,"Left",1,100000
GTNumber 0,2,150,60,60,20,"Right",2,100000
GTNumber 0,3,150,100,60,20,"Above",4,100000
GTNumber 0,4,150,140,60,20,"Below",8,100000
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

```
; *** FLAGS example 2
; *** Filename - FLAGS2.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTMX 0,1,150,80,60,20,"Disabled",$40,"Blitz|Amiga"
GTCheckBox 0,2,150,110,60,20,"On",$100
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.2 Numeric gadgets

This statement creates a string gadget which only allows numbers to be entered by the user.

GTINTEGER

```
Mode(s):   Amiga
Statement: create a number string gadget
Syntax:    GTInteger GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,DEFAULT
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. DEFAULT is the number that will initially appear in the box. Try the following example:

```
; *** GTInteger example
; *** Filename - GTInteger.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTInteger 0,1,150,80,60,20,"Enter a number",0,69
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTNUMBER

```
Mode(s):   Amiga
Statement: create a read-only number gadget
Syntax:    GTNumber GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,VALUE
```

The GTNUMBER statement creates a read-only numeric gadget.

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The VALUE parameter is the number to be displayed in the gadget:

```
; *** GTNumber example
; *** Filename - GTNumber.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTNumber 0,1,150,80,60,20,"Read only",0,100000
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTSETINTEGER

```
Mode(s):   Amiga
Statement: update contents of a GTInteger or GTNumber gadget
Syntax:    GTSetInteger GTLIST#,ID,VALUE
```

This statement updates and redraws the contents of a GTINTEGER or GTNUMBER gadget. VALUE is the new number to be displayed. Here's an example:

```
; *** GTSetInteger example
; *** Filename - GTSetInteger.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTInteger 0,1,150,80,60,20,"Enter a number",0,69
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
GTSetInteger 0,1,39
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTGETINTEGER

```
Mode(s):  Amiga
Function: return contents of a GTInteger or GTNumber gadget
Syntax:   i=GTGetInteger(GTLIST#,ID)
```

GTGETINTEGER returns the contents of a GTINTEGER or GTNUMBER gadget. For example:

```
; *** GTGetInteger example
; *** Filename - GTGetInteger.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTInteger 0,1,150,80,60,20,"Enter a number",0,69
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
A=GTGetInteger(0,1)
WLocate 10,120
NPrint "Your number is ",A
```

```
VWait 50
End
```

13.8.3 Text and string gadgets

GTBUTTON is used to create a text gadget. A text gadget is the simplest type of gadget, consisting of a sequence of characters (TEXT$) surrounded by a border.

GTBUTTON

```
Mode(s):   Amiga
Statement: create a text string gadget
Syntax:    GTButton GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. For example:

```
; *** GTButton example
; *** Filename - GTButton.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTButton 0,1,100,10,90,20," Click me ",0
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTSTRING

```
Mode(s):   Amiga
Statement: create a string gadget
Syntax:    GTString GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,CHARACTERS
```

The GTSTRING statement creates an Intuition-style "text input" gadget. When a string gadget is selected, a text cursor appears and characters may be input into the gadget, from the keyboard.

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The CHARACTERS parameter specifies the maximum number of characters that can be entered by the user. Try the following example:

```
; *** GTString example
; *** Filename - GTString.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTString 0,1,100,80,100,20,"Name:",0,11
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTTEXT

```
Mode(s):   Amiga
Statement: create a read-only text gadget
Syntax:    GTText GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,DISPLAY$
```

The GTTEXT statement creates a read-only text gadget.

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The DISPLAY$ parameter specifies a text string to display in the gadget:

```
; *** GTText example
; *** Filename - GTText.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTText 0,1,100,80,150,30,"Read only:",0,"This is a message"
Window 0,0,20,320,200,0,"Howdy Partner!",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTSETSTRING

```
Mode(s):   Amiga
Statement: update contents of a GTString or GTText gadget
Syntax:    GTSetString GTLIST#,ID,TEXT$
```

This statement updates the contents of a GTSTRING or GTTEXT gadget. The TEXT$ parameter specifies the new contents of the gadget. Example:

```
; *** GTSetString example
; *** Filename - GTSetString.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTString 0,1,100,80,100,20,"Name:",0,11
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
GTSetString 0,1,"Neil Wright"
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTGETSTRING

```
Mode(s):  Amiga
Function: return contents of a GTString or GTText gadget
Syntax:   s$=GTGetString(GTLIST#,ID)
```

The GTGETSTRING function returns the contents of a GTSTRING or GTTEXT gadget. Here is an example:

```
; *** GTGetString example
; *** Filename - GTGetString.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTString 0,1,100,80,100,20,"Name:",0,11
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
A$=GTGetString(0,1)
WLocate 10,120
NPrint "Your name is ",A$
```

```
VWait 50
End
```

13.8.4 Check box gadgets

GTCHECKBOX creates an Intuition-style check box. A check box is a gadget which toggles between on (the box is filled with a tick) and off (the box is empty).

GTCHECKBOX

```
Mode(s):   Amiga
Statement: create a check box gadget
Syntax:    GTCheckBox GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget:

```
; *** GTCheckBox example
; *** Filename - GTCheckBox.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTCheckBox 0,1,110,80,60,20,"Click on me",0
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.5 Cycle gadgets

The GTCYCLE statement is used to create cycle gadgets, which offer the user a range of options.

GTCYCLE

```
Mode(s):   Amiga
Statement: create a cycle gadget
Syntax:    GTCycle GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,OPTIONS$
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The OPTION$ paramater should consist of a list of options seperated by the | character (located directly above the return key). Here is an example:

```
; *** GTCycle example
; *** Filename - GTCycle.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTCycle 0,1,50,50,90,20,"Cycle",0,"Blitz|Basic|Is|Tops"
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.6 List gadgets

The next statement creates a list gadget. This is a gadget which contains a list of options which may be highlighted by the user. These options must be contained in a string field of a Blitz Basic linked list. This string field must be the second field, the first being a word type.

GTLISTVIEW

```
Mode(s):   Amiga
Statement: create a list gadget
Syntax:    GTListView GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,LIST()
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. Example:

```
; *** GTListView example
; *** Filename - GTListView.bb2

NEWTYPE .LIST
  A.w
  B$
End NEWTYPE
; *** First field (word type)
Dim List EMPTY.LIST(1000)
; *** Second field (string field)
Dim List GADGET.LIST(10)
While AddItem(GADGET()):GADGET()\B="Item #"+Str$(I)
  Let I+1
Wend
Screen 0,3+8
```

```
GTButton 0,0,10,10,60,20," QUIT ",0
GTListView 0,0,100,60,98,36,"GTListView",0,GADGET()
Window 0,0,20,320,200,$1000,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTCHANGELIST

```
Mode(s):   Amiga
Statement: modify GTListView list
Syntax:    GTChangeList GTLIST#,ID[,LIST()]
```

This statement is used to modify a GTLISTVIEW list. The correct procedure is as follows:

1. Execute GTCHANGELIST with no LIST() parameter.
2. Modify list.
3. Attach list with GTCHANGELIST and LIST() parameter.

For example:

```
; *** GTChangeList example
; *** Filename - GTChangeList.bb2

NEWTYPE .LIST
  A.w
  B$
End NEWTYPE
; *** First field (word type)
Dim List EMPTY.LIST(1000)
; *** Second field (string field)
Dim List GADGET.LIST(10)
While AddItem(GADGET()):GADGET()\B="Item #"+Str$(I)
  Let I+1
Wend
Screen 0,3+8
GTButton 0,0,10,10,60,20," QUIT ",0
GTListView 0,0,100,60,98,36,"GTListView",0,GADGET()
Window 0,0,20,320,200,$1000,"Window",1,0
AttachGTList 0,0
; *** Halve linked list
GTChangeList 0,0
For A=1 To 5
  KillItem GADGET()
Next A
; *** Add linked list
GTChangeList 0,0,GADGET()
```

```
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.7 Highlight gadgets

GTMX creates an exclusive selection gadget. This is a gadget which consists of a number of options which may be highlighted.

GTMX

```
Mode(s):   Amiga
Statement: create highlight gadget
Syntax:    GTMX GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,OPTIONS$
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The OPTION$ paramater should consist of a list of options seperated by the | character (located directly above the return key). For example:

```
; *** GTMX example
; *** Filename - GTMX.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTMX 0,1,150,80,60,20,"",0,"Blitz|Amiga"
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.8 Palette gadgets

This statement is used to create a number of coloured box gadgets. One of its main uses is as a palette selector.

GTPALETTE

```
Mode(s):   Amiga
Statement: create colour box gadget
Syntax:    GTPalette GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,DEPTH
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The DEPTH parameter specifies the number of box gadgets. For example:

```
; *** GTPalette example
; *** Filename - GTPalette.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTPalette 0,1,150,80,100,80,"Palette",0,5
Window 0,0,20,320,200,0,"Different",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.9 Proportional gadgets

As explained, proportional gadgets are "slider bars", such as the palette requesters in Electronic Art's excellent Deluxe Paint series.

GTSCROLLER

```
Mode(s):   Amiga
Statement: create a proportional gadget
Syntax:    GTScroller GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,VISIBLE,TOTAL
```

This statement creates a proportional gadget.

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The VISIBLE and TOTAL parameters specify the amount of visible and total amount of data to scroll through. For example:

```
; *** GTScroller example
; *** Filename - GTScroller.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTScroller 0,1,150,80,60,20,"Move me",0,5,200,100
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
```

```
Until GadgetHit=0
End
```

GTSLIDER

```
Mode(s):   Amiga
Statement: create a proportional gadget
Syntax:    GTSlider GTLIST#,ID,X,Y,W,H,TEXT$,FLAGS,MIN,MAX
```

This statement creates a proportional gadget for controlling the position of the display inside a larger view.

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. The W and H parameters specify the width and height of the gadget respectively, again in pixels. TEXT$ is a text string, or title, for the gadget. The MIN and MAX parameters specify the minimum and maximum amount of data to scroll through. For example:

```
; *** GTSlider example
; *** Filename - GTSlider.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTSlider 0,1,150,80,60,20,"Move me",0,0,5
Window 0,0,20,320,200,0,"A Window",1,0
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.10 Shape gadgets

The GTSHAPE statement is used to create gadgets with graphic elements, taken from a previously initialised shape bank.

GTSHAPE

```
Mode(s):   Amiga
Statement: create a shape gadget
Syntax:    GTShape GTLIST#,ID,X,Y,FLAGS,SHAPE1#[,SHAPE2#]
```

The X and Y parameters are the co-ordinates of the top-left of the gadget, relative to the window it is "attached" to, in pixels. If the optional SHAPE2# parameter is included then an alternative shape may be displayed when the gadget is selected. Try the following example:

```
; *** GTShape example
; *** Filename - GTShape.bb2

Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
Boxf 30,30,60,60,3
GetaShape 0,30,30,60,60
Boxf 30,30,60,60,1
Locate 4,5
NPrint "Hi!"
GetaShape 1,30,30,60,60
GTButton 0,0,10,10,60,20," QUIT ",0
GTShape 0,1,190,80,0,0,1
Window 0,0,20,320,200,0,"Window",1,0
WLocate 10,90
NPrint "Click on square shape:"
AttachGTList 0,0
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

13.8.11 Gadget borders

GTBEVELBOX

```
Mode(s):   Amiga
Statement: create border in current window
Syntax:    GTBevelBox GTLIST#,X,Y,WIDTH,HEIGHT,FLAGS
```

This statement draws a rectangular border into the currently used window. The X and Y parameters are the co-ordinates for the top left-hand corner of the border, and WIDTH and HEIGHT specify the width and height, in pixels, of the border. Here is a full example:

```
; *** GTBevelBox example
; *** Filename - GTBevelBox.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
GTBevelBox 0,50,50,100,100,0
Repeat
  ev.l=WaitEvent
```

```
Until GadgetHit=0
End
```

13.8.12 Manipulating gadgets

GadTools gadgets, like Blitz gadgets, can be altered or updated during runtime.

GTTOGGLE

```
Mode(s):   Amiga
Statement: toggle a toggle-type gadget
Syntax:    GTToggle GTLIST#,ID[,On/Off]
```

This statement toggles a toggle-type gadget (On) or (Off). For example:

```
; *** GTToggle example
; *** Filename - GTToggle.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTCheckBox 0,1,110,80,60,20,"Click on me",0
Window 0,0,20,320,200,0,"Window",1,0
AttachGTList 0,0
For A=1 To 20
  VWait 20
  GTToggle 0,1
  Redraw 0,1
Next A
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTDISABLE

```
Mode(s):   Amiga
Statement: disable a gadget
Syntax:    GTDisable GTLIST#,ID
```

GTENABLE

```
Mode(s):   Amiga
Statement: enable a gadget
Syntax:    GTEnable GTLIST#,ID
```

These two statements can disable and enable gadgets respectively. If a gadget is disabled then it is covered by a "mesh" and cannot be accessed by the user. Once a gadget is enabled, this mesh is removed and the gadget can be accessed. Example:

```
; *** GTDisable/GTEnable example
; *** Filename - GTDisable.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
GTButton 0,1,130,80,120,20,"Disable/Enable",0
Window 0,0,20,320,200,0,"Window",1,0
GTDisable 0,1
AttachGTList 0,0
VWait 100
GTEnable 0,1
Redraw 0,1
Repeat
  ev.l=WaitEvent
Until GadgetHit=0
End
```

GTGADPTR

```
Mode(s):  Amiga
Function: return location of GadTools gadget in memory
Syntax:   g=GTGadPtr(GTLIST#,ID)
```

This function returns the exact location of a GadTools gadget in memory:

```
; *** GTGadPtr example
; *** Filename - GTGadPtr.bb2

Screen 0,3
GTButton 0,0,10,10,60,20," QUIT ",0
Window 0,0,20,320,200,0,"A Window",1,0
AttachGTList 0,0
WLocate 5,80
Print "Location of gadget : ",GTGadPtr(0,0)
```

```
MouseWait
End
```

13.9 End-of-Chapter summary

A text gadget is the simplest type of gadget, consisting of a sequence of characters surrounded by an optional border.

The **STRINGGADGET** statement is used to create an Intuition "text input" gadget.

The SHAPEGADGET statement is used to create gadgets with graphic elements.

Proportional gadgets, or "slider bars", are the gadgets that are used to scroll the contents of Workbench windows around.

The GadTools library allows you to create more functional interfaces, involving list gadgets, cycle gadgets and highlight gadgets.

All gadgets can be enabled and disabled.

Chapter 14 : AGA

14.1 Advanced Graphics Architecture

The Amiga 1200 and 4000 series of computers are built around the AGA (Advanced Graphics Architecture) custom chipset, which supersedes the chipsets of earlier Amigas in graphics capability. The AGA (AA) chipset supports two to 256 colour register modes in resolutions from 320*200 to 1280*400 pixels. All colour display modes can display up to 256 colours from a palette of 16.8 million. Sprites can be displayed in high resolution and up to four times larger and the new HAM-8 mode, available in any display resolution, can display over 256,000 colours at once, from the 16.8 million colour palette.

Naturally, Blitz Basic 2 can take full control over the AGA chipset with the new Display Library. The library is an alternative to the Slice series of commands (consult the Chapter 6 for further information). Instead of simply extending the Slice library to support the new chipset, those clever people at Acid Software chose to develop a completely new set of commands.

Besides support for extended sprites, super hi-resolution scrolling and up to eight bitplanes, a more modular method of creating displays has been implemented with the use of CopLists. CopLists need only be initialised once at the start of the program. Displays can then be created using and combination of CopLists and, most importantly, the CreateDisplay command does not allocate any memory, thus avoiding any memory fragmenting problems.

Yes, I do know that this chapter is entitled "AGA", but the new display library is also semi-compatible with pre-AGA Amigas. CopLists can be created on ECS Amigas, however none of the AGA-specific flags (i.e. super high-resolution or 128/256 colours), nor any of the AGA-specific commands may be used on pre-AGA Amigas. You have been warned!

14.2 Creating a CopList

INITCOPLIST

```
Mode(s):   Amiga/Blitz
Statement: create a CopList
Syntax:    InitCopList LIST#,Y,H,FLAGS,SPRITES,COLS,CUSTOM
```

The INITCOPLIST statement, as its name implies, is used to create a CopList for use with the CREATEDISPLAY statement.

The Y and H parameters specify the vertical positioning and height of the CopList respectively. The Y parameter specifies the vertical location of the top of the CopList, ranging from 44 to the bottom of the current display. In other words, a value of 44 displays the CopList at the very top of the display.

The FLAGS parameter holds information on the number of colours, resolution etc. of the CopList (more on this later).

SPRITES specifies the number of sprites that can be displayed in the CopList (this should always be set to eight!).

The COLS parameter specifies the number of colours in the CopList, and the CUSTOM parameter should always be set to zero.

These are the FLAGS to be used with INITCOPLIST:

Table 14.1 : Screen Modes

Mode	Flag
Low-Res Mode	$00000
Hi-Res Mode	$00100
Super Hi-Res Mode	$00200

Table 14.2 : Screen Colours

Colours	Flag
2 Colours	$00001
4 Colours	$00002
8 Colours	$00003
16 Colours	$00004
32 Colours	$00005
64 Colours	$00006
128 Colours	$00007
256 Colours	$00008

Table 14.3 : New Flags

Option	Flag
24-Bit palette	$10000
Fetch Mode : Single	$00000
: Double	$01000
: Triple	$03000

Table 14.4 : Screen Options

Option	Flag
Smooth-Scroll	$00010
Dual-Playfield	$00020
Extra Half Bright	$00040
Hold And Modify	$00080

Table 14.5 : Sprite Modes

```
Mode                 Flag
===========================
Low-Res sprites      $00000
Hi-Res sprites       $00400
Super Hi-Res sprites $00800
```

To use more than one flag, you simply add them together using the "+" operator. This example would give you a 24-Bit palette, 256 colour CopList:

```
; *** InitCopList example
; *** Filename - InitCopList.bb2

; *** Pop into Blitz mode
BLITZ
; *** Open BitMap (256 colours)
BitMap 0,320,256,8
; *** Initialize CopList (256 colours)
InitCopList 0,44,256,$10000+$00008,8,256,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
MouseWait
End
```

Or:

```
; *** InitCopList example 2
; *** Filename - InitCopList2.bb2

BLITZ
BitMap 0,320,256,8
InitCopList 0,44,256,$10000+$03000+$00100+$00008,8,256,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
MouseWait
End
```

Which would give you a 24-Bit palette, triple fetch, hi-resolution, 256 colour CopList.

14.2.1 Multiple CopLists

You can also display multiple CopLists on a single BitMap. The following example demonstrates this:

```
; *** Multiple CopLists
; *** Filename - MultiCopList.bb2

; *** Open BitMap (256 colours)
BitMap 0,320,256,8
; *** Alter palette
For A=0 To 255
  AGAPalRGB 0,A,A,0,A
Next
; *** Open 2 CopLists
InitCopList 0,44,120,$13008,8,256,0
InitCopList 1,44+132,120,$13008,8,256,0
BLITZ
CreateDisplay 0,1
; *** Display CopList 0 in BitMap
DisplayBitMap 0,0
DisplayPalette 0,0
; *** Display CopList 1 in BitMap
DisplayBitMap 1,0
DisplayPalette 1,0
; *** Draw some BitMap graphics
For B=0 To 130
  Boxf B,B,320-B,255-B,B
Next
MouseWait
End
```

14.2.2 Non-AGA CopLists

The INITCOPLIST statement can also be used to create non-AGA displays, however only low-resolution sprites, single fetch mode, and a maximum of 64 colours are allowed:

```
; *** 2 colour CopList
; *** Filename - InitCopList3.bb2

BLITZ
BitMap 0,320,256,1
; *** Low-resolution 2 colour CopList
InitCopList 0,44,256,$00001,8,2,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
```

```
MouseWait
End
```

This is the same as the following SLICE example:

```
; *** Slice Vs CopList
; *** Filename - Sliced.bb2

BLITZ
BitMap 0,320,256,1
Slice 0,44,1
Show 0
MouseWait
End
```

CREATEDISPLAY

```
Mode(s):   Amiga/Blitz
Statement: setup screen display
Syntax:    CreateDisplay LIST#[,LIST#...]
```

CREATEDISPLAY is used to set up a new screen display with the new display library. Any number of CopLists can be passed to CREATEDISPLAY although at present they must be in order of vertical position and not overlap. CREATEDISPLAY then links the CopLists together using internal pointers. BitMaps, colours and sprites attached to CopLists are not affected. Here is a simple example:

```
; *** CreateDisplay example
; *** Filename - CreateDisplay.bb2

; *** Set colour 0 to black
AGAPalRGB 0,0,0,0,0
BLITZ
BitMap 0,320,256,8
; *** 256 colour, 24-Bit CopList
InitCopList 0,44,256,$10000+$00008,8,256,0
; *** Attach palette to display
DisplayPalette 0,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
MouseWait
End
```

14.3 Displaying a BitMap in a CopList

DISPLAYBITMAP

```
Mode(s):   Amiga/Blitz
Statement: show CopList
Syntax:    DisplayBitMap LIST#,BITMAP#[,X,Y][,BITMAP2#[X2,Y2]]
```

The DISPLAYBITMAP statement is used to display a BitMap in the specified CopList. If the optional X and Y parameters are included then the BitMap is positioned at these co-ordinates. Example:

```
; *** DisplayBitmap example
; *** Filename - DisplayBitmap.bb2

For COL=0 To 254
  AGAPalRGB 0,COL,COL,COL,COL
Next COL
BLITZ
BitMap 0,320,256,8
InitCopList 0,44,256,$10000+$00008,8,256,0
; *** Set up screen display
CreateDisplay 0
; *** Attach palette to display
DisplayPalette 0,0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
For A=0 To 200
  Circle Rnd(320),Rnd(250),Rnd(30)+10,A
Next A
MouseWait
End
```

If the BitMap is physically larger than the CopList then the DISPLAYBITMAP statement may be used to scroll the BitMap about the display.

With AGA machines, the X positioning of low-resolution and hi-resolution CopLists uses the fractional part of the X parameter for super-smooth scrolling. The CopList must be initialised with the smooth-scrolling flag ($00010) set if the X and Y parameters are used.

For dual-playfield CopLists (flag $00020), if the optional BITMAP2# parameter is included then a second BitMap is displayed in the background of the CopList (the first BitMap (BITMAP#) is displayed in the foreground). If the optional X2 and Y2 parameters are included then the background BitMap is positioned at these co-ordinates.

The following example uses DISPLAYBITMAP to scroll an AGA starfield:

```
; *** DisplayBitMap example 2
; *** Filename - DisplayBitMap2.bb2

; *** Create colour palette
For A=0 To 255
  AGAPalRGB 0,A,A,0,0
Next A
BLITZ
BitMap 0,640,256,8
; *** 256 colour, 24-Bit, smooth-scroll CopList
InitCopList 0,44,256,$10000+$00008+$00010,8,256,0
; *** Attach palette to display
DisplayPalette 0,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
; *** Plot starfield
For B=0 To 2500
  Plot Rnd(640),Rnd(250),Rnd(254)+1
Next B
; *** Scroll BitMap
Repeat
  VWait
  DisplayBitMap 0,0,X,0
  X=QWrap(X+1,0,640)
Until Joyb(0)>0
MouseWait
End
```

14.4 Palettes

Palette objects, or palettes, are temporary storage areas of colour information. This information can be taken either from an IFF file or created from scratch.

The LOADPALETTE statement can be used to load a palette object from disk (consult Chapter 7 for more information).

If colour information is created by the user then it will not affect the current display colours until the DISPLAYPALETTE statement has been executed.

DISPLAYPALETTE

```
Mode(s):   Amiga/Blitz
Statement: copy colour information to CopList
Syntax:    DisplayPalette LIST#,PALETTE#[,COLOUR_OFFSET]
```

This statement copies colour information from a Palette to the CopList specified. It is similar in usage to the USE PALETTE statement. Here is an example:

```
; *** 256 colour CopList
; *** Filename - DisplayPalette.bb2

BLITZ
; *** Define palette
For A=0 To 255
  AGAPalRGB 0,A,A,0,0
Next A
BitMap 0,320,256,8
; *** 256 colour, 24-Bit CopList
InitCopList 0,44,256,$10000+$00008,8,256,0
; *** Attach palette to display
DisplayPalette 0,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
; *** Create BitMap graphics
For A=0 To 1500
  Circlef Rnd(320),Rnd(256),Rnd(20)+1,Rnd(254)+1
Next A
MouseWait
End
```

AGAPALRGB

```
Mode(s):   Amiga/Blitz
Statement: set a colour register within a palette object
Syntax:    AGAPalRGB PALETTE#,REGISTER,RED,GREEN,BLUE
```

The AGAPALRGB statement allows you to set an individual colour register within a palette object. The colour change will not become evident until either of the USE PALETTE or DISPLAY PALETTE statements is used. Try the following example:

```
; *** AGAPalRGB example
; *** Filename - AGAPalRGB.bb2

BLITZ
; *** Define palette
For A=0 To 255
  AGAPalRGB 0,A,0,A,0
Next A
BitMap 0,320,256,8
; *** 256 colour, 24-Bit CopList
```

```
InitCopList 0,44,256,$10000+$00008,8,256,0
; *** Attach palette to display
DisplayPalette 0,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
; *** Draw BitMap graphics
For B=0 To 254
  X=Rnd(320)
  Y=Rnd(256)
  Boxf X,Y,X+Rnd(50),Y+Rnd(50),B
Next B
MouseWait
End
```

AGARGB

```
Mode(s):   Amiga
Statement: set a colour register to an RGB colour value
Syntax:    AGARGB REGISTER,RED,GREEN,BLUE
```

AGARGB allows you to set an individual colour register in a palette to an RGB colour value. AGARGB does not affect palette objects.

The AGARED, AGAGREEN and AGABLUE statements return the amount of their respected colour in a specified colour register. The returned values range from zero to 255.

AGARED

```
Mode(s):   Amiga
Function:  return the amount of RGB red in a colour register
Syntax:    r=AGARed(REGISTER)
```

AGAGREEN

```
Mode(s):   Amiga
Function:  return the amount of RGB green in a colour register
Syntax:    g=AGAGreen(REGISTER)
```

AGABLUE

```
Mode(s):   Amiga
Function:  return the amount of RGB blue in a colour register
Syntax:    b=AGABlue(REGISTER)
```

14.5 A full example

The following example demonstrates the correct procedure for displaying AGA graphics in Blitz Basic. Remember to replace the filename below with the name of an AGA file to load, otherwise the program will fail - I personally reccommend the Tutankhamun picture from DeluxePaint IV AGA:

```
; *** AGA display example
; *** Filename - AGA.bb2

BitMap 0,320,DispHeight,8
; *** Load AGA picture plus palette
LoadBitMap 0,"FILENAME.IFF"
LoadPalette 0,"FILENAME.PALETTE",0
; *** Create 256 colour CopList
InitCopList 0,34,200,$10418,8,256,0
BLITZ
Mouse On
MouseArea 0,0,320,DispHeight
; *** Set up screen display
CreateDisplay 0
; *** Attach palette to display
DisplayPalette 0,0
; *** Scroll BitMap with mouse
While Joyb(0)=0
  VWait
  DisplayBitMap 0,0,MouseX,0
Wend
End
```

14.6 AGA Sprite handling

AGA Sprites are initialised by either loading them from disk, or by converting a shape object into a sprite object using the GETASPRITE statement.

Sprites are handled entirely by the Amiga's hardware so they do not interfere or corrupt BitMap graphics in any way. AGA Sprites can be displayed in high-resolution and up to four times larger than normal!

DISPLAYSPRITE

```
Mode(s):   Blitz
Statement: position a sprite
Syntax:    DisplaySprite LIST#,SPRITE#,X,Y,CHANNEL
```

DISPLAYSPRITE is used to display a sprite in a CopList. LIST# is the number of the CopList to display the sprite on and SPRITE# is the sprite number (taken from a previously initialised sprite bank). The X and Y parameters specify the co-ordinates of the sprite, in pixels, and the CHANNEL parameter specifies the sprite channel to be used to display the sprite:

```
; *** DisplaySprite example
; *** Filename - DisplaySprite.bb2

BLITZ
; *** Define palette
For COLS=0 To 16
  AGAPalRGB 0,COLS,COLS,0,0
Next COLS
BitMap 0,320,256,4
; *** Make a sprite
For A=4 To 10
  Boxf 10+A*2,10+A*2,55-A*2,55-A*2,A
Next A
GetaShape 0,10,10,40,40
GetaSprite 0,0
Cls
; *** Create CopList
InitCopList 0,44,256,$00001,8,16,0
; *** Attach palette to display
DisplayPalette 0,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
; *** Move sprite
For X=1 To 260
  VWait
  DisplaySprite 0,0,X,50,0
Next X
End
```

SPRITEMODE

```
Mode(s):   Amiga/Blitz
Statement: define the width of sprites to be used
Syntax:    SpriteMode MODE
```

SPRITEMODE specifies the width of sprites to be used in a Blitz Basic program:

Table 14.6 : The MODE parameter

```
MODE  Sprite width
==================
0     16
1     32
2     64
```

For example:

```
; *** SpriteMode example
; *** Filename - SpriteMode.bb2

; *** Set sprite width to 64 pixels
SpriteMode 2
BLITZ
; *** Define palette
For COLS=0 To 16
  AGAPalRGB 0,COLS,COLS,0,0
Next COLS
BitMap 0,320,256,4
; *** Make a sprite
Boxf 0,0,64,64,6
GetaShape 0,0,0,64,64
GetaSprite 0,0
Cls
; *** Create CopList
InitCopList 0,44,256,$00001,8,16,0
; *** Attach palette to display
DisplayPalette 0,0
; *** Set up screen display
CreateDisplay 0
; *** Attach BitMap to CopList
DisplayBitMap 0,0
; *** Move sprite
For X=1 To 260
  VWait
  DisplaySprite 0,0,X,50,0
Next X
End
```

14.7 End-of-Chapter summary

The new display library is an alternative to the Slice commands. Slices are replaced by CopLists, a more modular method of creating displays.

CopLists can use from two to 256 colours from a palette of 16.8 million. CopLists can be created on ECS Amigas, however none of the AGA-specific flags (i.e. super hi-resolution or 128/256 colours), nor any of the AGA-specific commands may be used on pre-AGA Amigas.

The **DISPLAYBITMAP** statement can be used to scroll a super-BitMap (one larger than the physical display) around a CopList.

Sprites can be up to 64 pixels in width and displayed in high-resolution.

Chapter 15 : System Functions

This chapter will show you how to design software which can dynamically adjust itself to different displays (NTSC or PAL). It will show you how to obtain information about Blitz Basic objects, the system processor and the Workbench screen. You will also learn about the BREXX system.

15.1 Display heights

In the United Kingdom we have adopted the PAL (I) standard of television system. However, half of the world use the NTSC system. This is where the programmer's problems begin.

PAL updates 50 times a second (50Hz) and can display a maximum of 256 horizontal lines. NTSC, however, updates 60 times a second (60Hz), and can only display a maximum of 200 horizontal lines.

The update problem can be overcome by making sure that all time-essential routines can be executed in under a sixtieth of a second. If you don't make allowances for this then your programs will appear jerky on the other system. This is because, although the VWAIT statement can be used to tie-in screen syncronisation, its delay varies between PAL and NTSC systems.

The screen height problem is also relatively easy to work around. Say, for example, you have created a piece of software on a PAL system which uses a screen display of 320 by 256 pixels. Under the NTSC system the bottom part of the screen would be hidden from view! This is where the following two commands come in.

NTSC

```
Mode(s):  Amiga/Blitz
Function: identify NTSC or PAL machines
Syntax:   n=NTSC
```

The NTSC function is used to identify whether or not an NTSC machine is in use. The function returns (-1) if the display is currently in NTSC mode and (0) if the display is in PAL mode. This is primarily of use when designing software which can dynamically adjust itself to different displays. For example:

```
; *** NTSC example
; *** Filename - NTSC.bb2

If NTSC
  ; *** The good old U.S of A (probably!)
  NPrint "NTSC mode"
Else
  ; *** We're in England, mate!
  NPrint "PAL mode"
EndIf
MouseWait
End
```

DISPHEIGHT

```
Mode(s):  Amiga/BLitz
Function: return maximum available screen height
Syntax:   d=DispHeight
```

This function returns a value of 256 if it is executed in PAL mode, or 200 if executed on an NTSC Amiga:

```
; *** DispHeight example
; *** Filename - DispHeight.bb2

NPrint DispHeight," pixels high"
MouseWait
End
```

15.2 Object handling

Objects are structures designed to control multiple system elements, such as graphics, files and screens. Here is a list of the Blitz Basic 2 objects:

Table 15.1 : Blitz Basic objects

```
Object       Description
================================
BitMaps      Display elements
Blitzfonts   BitMap fonts
CopLists     Display elements
Files        File handling
GadgetLists  Intuition gadgets
Intuifonts   Intuition fonts
MenuLists    Intuition menus
Modules      Tracker music
Palettes     Palette information
Screens      Intuition screens
Shapes       Blitter objects
Sounds       Samples
Sprites      Hardware sprites
Windows      Intuition windows
```

All objects can be created, manipulated and destroyed, or deleted. The manipulation of the above objects has been covered in the previous chapters. Objects can also be controlled with the following commands.

USE

```
Mode(s):   Amiga/Blitz
Statement: set specified object as current object
Syntax:    Use NAME OBJECT#
```

The USE statement sets the current object (NAME) as number OBJECT#. NAME can be any object, such as Screen, Slice, BitMap, Window etc. The OBJECT# parameter specifies the object number. Try the following example:

```
; *** Use example
; *** Filename - Use.bb2

; *** Open a screen and grab its BitMap
Screen 0,3,"Back"
ScreensBitMap 0,0
; *** Enable text output
BitMapOutput 0
; *** Open screen at front of display
Screen 1,0,30,320,200,3,0,"Front",1,2
; *** Set current screen as 0
Use Screen 0
Locate 0,2
NPrint "I feel Used!"
MouseWait
End
```

FREE

```
Mode(s):   Amiga/Blitz
Statement: free the specified object
Syntax:    Free NAME OBJECT#
```

FREE is used to remove a specified object. All Blitz Basic objects can be removed with the FREE statement (e.g. Free BitMap 0). NAME can be any object, such as Screen, Slice, BitMap, Window etc. The OBJECT# parameter specifies the object number. Objects are automatically freed when a program ends. For example:

```
; *** Free example
; *** Filename - Free.bb2

; *** Open a simply screen
Screen 0,3,"Bye!"
VWait 100
```

```
; *** Remove screen from display
Free Screen 0
MouseWait
End
```

USED

```
Mode(s):  Amiga/Blitz
Function: return currently used object number
Syntax:   u=Used NAME
```

This function returns the currently used object number. NAME can be any object, such as Screen, Slice, BitMap, Window etc. Here's an example:

```
; *** Used example
; *** Filename - Used.bb2

; *** Nip into Blitz mode
BLITZ
; *** Open a Blitz mode display
BitMap 1,320,256,3
BitMapOutput 1
Slice 2,44,3
Show 1
; *** Return BitMap number (1)
NPrint "BitMap number = ",Used BitMap
; *** Return Slice number (2)
NPrint "Slice number  = ",Used Slice
MouseWait
End
```

ADDR

```
Mode(s):  Amiga/Blitz
Function: return address of object in memory
Syntax:   a=Addr NAME(OBJECT#)
```

The ADDR function returns the address of a particular object in memory. NAME can be any object, such as Screen, Slice, BitMap, Window etc. The OBJECT# parameter specifies the object number. Try this example:

```
; *** Addr example
; *** Filename - Addr.bb2

; *** Pop into Blitz mode
BLITZ
; *** Open Blitz mode display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
BitMapOutput 0
; *** Return address of Slice
NPrint "Slice object 0 is at: ",Addr Slice(0)
MouseWait
End
```

MAXIMUM

```
Mode(s):  Amiga/Blitz
Function: return maximum setting for an object
Syntax:   m=Maximum NAME
```

MAXIMUM reutrns the maximum setting for an object (i.e. the maximum number of the object allowed by Blitz). NAME can be any object, such as Screen, Slice, BitMap, Window etc. Maximum settings are entered into the "OPTIONS" requester of the "COMPILER" menu of the Blitz Basic editor. Example:

```
; *** Maximum example
; *** Filename - Maximum.bb2

NPrint "Maximum screens = ",Maximum Screen
NPrint "Maximum windows = ",Maximum Window
NPrint "Maximum menus   = ",Maximum Menus
NPrint "Maximum Slices  = ",Maximum Slice
MouseWait
End
```

15.3 System date and time

The following statements and functions can be used to read and manipulate the system date and time, as displayed on the Workbench screen.

SYSTEMDATE

```
Mode(s):  Amiga
Function: return the system date
Syntax:   s=SystemDate
```

The SYSTEMDATE function returns the system date as the number of days passed since 1/1/1978. For example:

```
; *** SystemDate example
; *** Filename - SystemDate.bb2

NPrint SystemDate," days passed"
MouseWait
End
```

DATE$

```
Mode(s):  Amiga
Function: return system date (formatted)
Syntax:   d$=Date$(DAYS)
```

DATE$ converts the format returned by SystemDate (days passed since 1/1/1978) into a string format of dd/mm/yyyy or mm/dd/yyyy depending on the date format (defaults to 0). Here is an example:

```
; *** Date$ example
; *** Filename - Date$.bb2

NPrint Date$(SystemDate)
MouseWait
End
```

NUMDAYS

```
Mode(s):  Amiga
Function: return system date (day count)
Syntax:   n=NumDays(DATE$)
```

This function converts a string created using DATE$ to the relevant day count since 1/1/1978. Example:

```
; *** NumDays example
; *** Filename - NumDays.bb2

; *** Return date
A=SystemDate
NPrint A
; *** Format date
B$=Date$(A)
NPrint B$
; *** Return day count
C=NumDays(B$)
NPrint C
MouseWait
End
```

DATEFORMAT

```
Mode(s):   Amiga
Statement: format string representation
Syntax:    DateFormat FORMAT#
```

The DATEFORMAT statement is used to configure the way both DAY$ and NUMDAYS output. The FORMAT# parameter can be either (0) or (1):

Table 15.2 : Date format

FORMAT#	Output	Example
0	dd/mm/yyyy	28/03/1978
1	mm/dd/yyyy	03/28/1978

Here is an example:

```
; *** DateFormat example
; *** Filename - DateFormat.bb2

A=SystemDate
B$=Date$(A)
; *** dd/mm/yyyy format (e.g. 25/12/1994)
DateFormat 0
NPrint B$
; *** mm/dd/yyyy format (e.g. 01/31/2001)
DateFormat 1
NPrint B$
```

```
MouseWait
End
```

The following functions return the hours, minutes, seconds, days, months and years when SYSTEMDATE was last called respectively.

HOURS

```
Mode(s):  Amiga
Function: return hours relevant to SystemDate
Syntax:   h=Hours
```

Example:

```
; *** Hours example
; *** Filename - Hours.bb2

NPrint Date$(SystemDate)
NPrint Hours
MouseWait
End
```

MINS

```
Mode(s):  Amiga
Function: return minutes relevant to SystemDate
Syntax:   m=Mins
```

Example:

```
; *** Mins example
; *** Filename - Mins.bb2

NPrint Date$(SystemDate)
NPrint Mins," minutes"
MouseWait
End
```

SECS

```
Mode(s):  Amiga
Function: return seconds relevant to SystemDate
Syntax:   s=Secs
```

Example:

```
; *** Secs example
; *** Filename - Secs.bb2

NPrint Date$(SystemDate)
For A=1 To 10
  A$=Date$(SystemDate)
  NPrint Secs," seconds"
  VWait 50
Next A
End
```

DAYS

```
Mode(s):  Amiga
Function: return day relevant to SystemDate
Syntax:   d=Days
```

Example:

```
; *** Days example
; *** Filename - Days.bb2

NPrint Date$(SystemDate)
NPrint Days
MouseWait
End
```

MONTHS

```
Mode(s):  Amiga
Function: return months relevant to SystemDate
Syntax:   m=Months
```

Example:

```
; *** Months example
; *** Filename - Months.bb2

NPrint Date$(SystemDate)
NPrint Months
MouseWait
End
```

YEARS

```
Mode(s):  Amiga
Function: return years relevant to SystemDate
Syntax:   y=Years
```

Example:

```
; *** Years example
; *** Filename - Years.bb2

NPrint Date$(SystemDate)
NPrint Years
MouseWait
End
```

WEEKDAY

```
Mode(s):  Amiga
Function: return weekday relevant to SystemDate
Syntax:   w=WeekDay
```

The WEEKDAY function returns the weekday relevant to the last call to SYSTEMDATE.

Table 15.3 : Values returned by WEEKDAY

```
Value  Weekday
================
0      Sunday
1      Monday
2      Tuesday
3      Wednesday
4      Thursday
```

```
5      Friday
6      Saturday
```

Try the following example:

```
; *** WeekDay example
; *** Filename - WeekDay.bb2

NPrint Date$(SystemDate)
NPrint WeekDay
MouseWait
End
```

Here is a full example which demonstrates the above statements and functions:

```
; *** Got the time?
; *** Filename - SystemDate.bb2

; *** Arrays to hold day and month strings
Dim D$(6),M$(12)
; *** Read day and month data into arrays
Restore DAYNAMES
For A=0 To 6
  Read D$(A)
Next A
Restore MONTHNAMES
For B=1 To 12
  Read M$(B)
Next B
; *** Output system date and time
NPrint Date$(SystemDate)
NPrint D$(WeekDay)," ",Days," ",M$(Months)," ",Years
NPrint Hours,":",Mins,":",Secs
MouseWait
End

; *** Day data
DAY:
Data$ SUNDAY,MONDAY,TUESDAY,WEDNESDAY
Data$ THURSDAY,FRIDAY,SATURDAY

; *** Month data
MONTH:
Data$ JAN,FEB,MAR,APR,MAY,JUN,JUL
Data$ AUG,SEP,OCT,NOV,DEC
```

15.4 Workbench functions

The following two functions return the width and height of the current Workbench screen, in pixels, respectively.

WBWIDTH

```
Mode(s):  Amiga
Function: return the width of Workbench screen
Syntax:   w=WBWidth
```

Example:

```
; *** WBWidth example
; *** Filename - WBWidth.bb2

X=WBWidth
NPrint "Workbench is:"
NPrint ""
NPrint X," pixels wide"
MouseWait
End
```

WBHEIGHT

```
Mode(s):  Amiga
Function: return the height of Workbench screen
Syntax:   h=WBHeight
```

For example:

```
; *** WBHeight example
; *** Filename - WBHeight.bb2

Y=WBHeight
NPrint "Workbench is:"
NPrint ""
NPrint Y," pixels high"
MouseWait
End
```

WBDEPTH

```
Mode(s):  Amiga
Function: return the depth of Workbench screen
Syntax:   d=WBDepth
```

WBDEPTH returns the depth of the current Workbench screen, in bitplanes. Here is an example:

```
; *** WBDepth
; *** Filename - WBDepth.bb2

D=WBDepth
NPrint "Workbench is:"
NPrint ""
NPrint D," bitplanes"
NPrint 2^D," colours"
MouseWait
End
```

WBVIEWMODE

```
Mode(s):  Amiga
Function: return the viewmode of Workbench screen
Syntax:   v=WBViewMode
```

This statement returns the viewmode of the current Workbench screen. The different values of WBViewMode are as follows:

Table 15.4 : Values returned by WBVIEWMODE

```
Value         Description
=========================
32768 ($8000) Hires
4     ($0004) Interlace
0     ($0000) Lowres
```

For example:

```
; *** WBViewMode example
; *** Filename - WBViewMode.bb2

NPrint Abs(WBViewMode)
```

```
MouseWait
End
```

15.5 Food processor

The Central Processing Unit, or CPU, lies at the heart of the Amiga. All Amigas are powered by a Motorola chip, from the 68000 (Amiga 500), to the 68040 (Amiga 4000). You may find it useful, at times, to find out the processor of the "host" computer. This is primarily of use when designing maths-intensive software which can take advantage of a faster processor.

PROCESSOR

```
Mode(s):  Amiga
Function: return the system processor number
Syntax:   p=Processor
```

This handly little function returns a value which indicates the system processor number (the chip inside the Amiga that your program is running on).

Table 15.5 : Values returned by PROCESSOR

Value	Processor
0	68000
1	68010
2	68020
3	68030
4	68040

For example, if zero is returned then your system processor is a 68000 microprocessor, and if four is returned then you are lucky enough to own a 68040 processor. Here is a full example:

```
; *** Processor example
; *** Filename - Processor.bb2

A=Processor
Print "Your processor is a 680",A,"0"
MouseWait
End
```

EXECVERSION

```
Mode(s):  Amiga
Function: return the Exec version number
Syntax:   e=ExecVersion
```

Similarly, EXECVERSION returns a value which indicates the Operating System number:

Table 15.6 : Values returned by EXECVERSION

```
Value  Operating system  Amiga
=========================================
33     1.2               A1000/A2000
34     1.3               A500/A2000/A3000
36     2.0               A500P/A600
39     3.0               A1200/A4000
```

Try the following example:

```
; *** ExecVersion example
; *** Filename - ExecVersion.bb2

A=ExecVersion
Print "Your EXEC version is ",A
MouseWait
End
```

15.6 BREXX

BREXX describes a set of Blitz Basic commands which can be used to simulate user input, such as mouse movements and keyboard input.

The BREXX system also allows you to create tape objects. Tape objects are predefined sequences of events which can be played back at any time, independently of the main program. Each tape object can store the movement of the mouse pointer, the status of the mouse buttons, or keyboard input. None of the BREXX commands are available in Blitz mode.

15.6.1 Emulating user input

The following commands can be used to move the mouse pointer, alter the status of the mouse buttons and emulate keyboard input, without the user having touched the keyboard or mouse.

ABSMOUSE

```
Mode(s):   Amiga
Statement: move mouse pointer
Syntax:    AbsMouse X,Y
```

The ABSMOUSE statement is used to position the mouse pointer at an absolute display location. The X and Y parameters specify the new co-ordinates of the pointer. X must be in the range zero to 639 and Y must be in the range zero to 399 (NTSC) or zero to 511 (PAL). Here's an example:

```
; *** AbsMouse example
; *** Filename - AbsMouse.bb2

; *** Position mouse at top-left of screen
AbsMouse 0,0
VWait 30
; *** Position mouse in middle of screen
AbsMouse 319,199
VWait 30
; *** Position mouse at bottom-right of screen
AbsMouse 630,390
MouseWait
End
```

RELMOUSE

```
Mode(s):   Amiga
Statement: move mouse pointer
Syntax:    RelMouse XOFFSET,YOFFSET
```

RELMOUSE is similar to ABSMOUSE, except is moves the mouse pointer a relative distance from its current position. If the XOFFSET parameter is positive then the pointer will be moved rightwards, and if it is negative then the pointer will be moved leftwards. If YOFFSET is positive then the pointer is moved downwards, and if it is negative then the pointer is moved upwards. For example:

```
; *** RelMouse example
; *** Filename - RelMouse.bb2

; *** Position mouse near middle of display
AbsMouse 300,100
; *** Move mouse rightwards
For X=1 To 40
  RelMouse 6,0
Next X
```

```
; *** Wait for a mouse click
MouseWait
End
```

MOUSEBUTTON

```
Mode(s):   Amiga
Statement: alter status of mouse button
Syntax:    MouseButton BUTTON,On/Off
```

This statement is used to alter the status of either mouse button. The BUTTON parameter should be set to (0) for the left mouse button and (1) for the right mouse button. If a button is set to "On" then it is pressed, and if it is set "Off" then it is released. Example:

```
; *** MouseButton example
; *** Filename - MouseButton.bb2

For A=1 To 10
  VWait 2
  ; *** Toggle right mouse button on
  MouseButton 1,On
  VWait 2
  ; *** Toggle right mouse button off
  MouseButton 1,Off
Next A
End
```

CLICKBUTTON

```
Mode(s):   Amiga
Statement: alter status of mouse button
Syntax:    ClickButton BUTTON
```

CLICKBUTTON works similarly to MOUSEBUTTON, except it presses and releases the specified mouse button. The BUTTON parameter should be set to (0) for the left mouse button and (1) for the right mouse button. Try this example:

```
; *** ClickButton example
; *** Filename - ClickButton.bb2

; *** Open a standard text gadget
TextGadget 0,32,32,0,1," Magic gadget "
; *** Open screen and window for gadget
Screen 0,3
```

```
Window 0,0,0,320,200,$100f,"Window",1,2,0
; *** Position mouse pointer at top of display
AbsMouse 0,70
; *** Move mouse pointer to gadget
For A=1 To 30
  RelMouse 4,0
Next A
; *** Highlight gadget
ClickButton 0 MouseWait End
```

TYPE

```
Mode(s):   Amiga
Statement: output a text string character by character
Syntax:    Type TEXT$
```

The TYPE statement outputs a text string to the screen, character by character. This generates a "typewriter-style" effect. For example:

```
; *** Type example
; *** Filename - Type.bb2

NPrint "Input your name:-"
; *** Input some text
NAME$=Edit$(20)
; *** Type text string, character by character
Type "Your name is "+NAME$
MouseWait
End
```

15.6.2 Recording tape objects

RECORD

```
Mode(s):   Amiga
Statement: record a tape object
Syntax:    Record [TAPE#]
```

This statement allows you to record a tape object. Tabe objects are sequences of mouse movements and/or keyboard events which may be played back at any time.

To record a tape with RECORD, the optional TAPE# parameter must be included:

```
Record 0 ; *** Record tape 0
```

To finish recording, the RECORD statement with no parameters should be used instead:

```
Record ; *** Stop recording
```

Here is an example:

```
; *** Record example
; *** Filename - Record.bb2

NPrint "Move the mouse then press the right button"
; *** Position mouse pointer at top-left of display
AbsMouse 0,0
; *** Start recording mouse movements
Record 0
; *** Record until right mouse button pressed
While Joyb(0)<>2
Wend
; *** Stop recording
Record
; *** Position mouse at top-left again
AbsMouse 0,0
; *** Play tape
PlayBack 0
MouseWait
End
```

15.6.3 Playing tape objects

PLAYBACK

```
Mode(s):   Amiga
Statement: play a tape object
Syntax:    PlayBack [TAPE#]
```

The PLAYBACK statement plays a previously initialised tape object. To play a tape the optional TAPE# parameter must be included:

```
PlayBack 0 ; *** Play tape 0
```

To finish playing, the PLAYBACK statement with no parameters should be used instead:

```
PlayBack ; *** Stop tape
```

Try the following example:

```
; *** PlayBack example
; *** Filename - PlayBack.bb2

NPrint "Move the mouse then press the right button"
; *** Position mouse at top-left
AbsMouse 0,0
; *** Start recording mouse movements
Record 0
While Joyb(0)<>2
Wend
; *** Stop recording
Record
; *** Position mouse at top-left again
AbsMouse 0,0
; *** Play tape
PlayBack 0
MouseWait
End
```

QUICKPLAY

```
Mode(s):   Amiga
Statement: toggle PlayBack execution mode
Syntax:    QuickPlay On/Off
```

QUICKPLAY is used to alter the way in which tape objects are played by the PLAYBACK statement. If QUICKPLAY is set to (On) then tapes will be played with no delays between actions (any pauses will be ignored). If QUICKPLAY is set to (Off) then the pauses will be included - this is the default mode. For example:

```
; *** QuickPlay example
; *** Filename - QuickPlay.bb2

NPrint "Move the mouse then press the right button"
AbsMouse 0,0
; *** Start recording mouse movements
Record 0
While Joyb(0)<>2
Wend
```

```
; *** Stop recording
Record
AbsMouse 0,0
; *** Toggle QuickPlay on (no delays)
QuickPlay On
; *** Play tape
PlayBack 0
MouseWait
End
```

PLAYWAIT

```
Mode(s):   Amiga
Statement: pause program execution
Syntax:    PlayWait(TAPE#)
```

PLAYWAIT pauses program execution until the PLAYBACK statement has finished playing a tape. The TAPE# parameter should be set to the current tape object. Example:

```
; *** PlayWait example
; *** Filename - PlayWait.bb2

AbsMouse 0,0
; *** Record mouse movements
Record 0
While Joyb(0)<>2
Wend
; *** Stop recording
Record
AbsMouse 0,0
NPrint "Program execution stopped"
PlayBack 0
; *** Halt program execution
PlayWait(0)
NPrint "Program execution started"
MouseWait
End
```

15.6.4 BREXX functions

XSTATUS

```
Mode(s):  Amiga
Function: return status of BREXX system
Syntax:   x=XStatus
```

This function returns the status of the BREXX system.

Table 15.7 : XStatus return values

Value	Description
0	BREXX is currently inactive
1	BREXX is recording a tape
2	BREXX is playing a tape

Here's an example:

```
; *** XStatus example
; *** Filename - XStatus.bb2

; *** Open a screen and grab its BitMap
Screen 0,11
ScreensBitMap 0,0
BitMapOutput 0
Locate 0,3
NPrint "Move the mouse then press the right button"
AbsMouse 0,0
Record 0
VWait 20
Locate 0,6
; *** BREXX is recording
NPrint XStatus," - recording !!!!"
While Joyb(0)=0
Wend
Record
Locate 0,6
; *** BREXX is inactive
NPrint XStatus," - inactive !!!! "
VWait 100
AbsMouse 0,0
PlayBack 0
VWait 20
Locate 0,6
; *** BREXX is playing
NPrint XStatus," - playing !!!!  "
MouseWait
End
```

15.6.5 Loading and saving tape objects

LOADTAPE

```
Mode(s):   Amiga
Statement: load a tape object
Syntax:    LoadTape TAPE#,"FILENAME"
```

This statement is used to load a tape object into memory. For example:

```
; *** LoadTape example
; *** Filename - LoadTape.bb2

NPrint "Move the mouse then press the right button"
AbsMouse 0,0
; *** Create a tape object and save to RAM:
Record 0
While Joyb(0)<>2
Wend
Record
SaveTape 0,"RAM:TAPE"
VWait 20
; *** Load tape from RAM:
LoadTape 0,"RAM:TAPE"
AbsMouse 0,0
; *** Play tape
PlayBack 0
MouseWait
End
```

SAVETAPE

```
Mode(s):   Amiga
Statement: save a tape object
Syntax:    SaveTape TAPE#,"FILENAME"
```

This statement saves a tape object to disk. For example:

```
; *** SaveTape example
; *** Filename - SaveTape.bb2

NPrint "Move the mouse then press the right button"
; *** Create a tape object
AbsMouse 0,0
Record 0
```

```
While Joyb(0)<>2
Wend
Record
; *** Save tape to RAM:
SaveTape 0,"RAM:TAPE"
End
```

15.6.6 Recording BREXX commands

The following commands are also used to record tape objects. However, TAPETRAP and QUIETTRAP are used to record sequences of BREXX commands, and not mouse movements or keyboard events.

TAPETRAP

```
Mode(s):   Amiga
Statement: record a sequence of BREXX commands
Syntax:    TapeTrap [TAPE#]
```

TAPETRAP is used to record a sequence of BREXX commands (ABSMOUSE, RELMOUSE, MOUSEBUTTON or CLICKBUTTON). To record a tape the optional TAPE# parameter must be included:

```
TapeTrap 0 ; *** Record tape 0
```

To finish recording, the TAPETRAP statement with no parameters should be used instead:

```
TapeTrap ; *** Stop tape
```

Try this example:

```
; *** TapeTrap example
; *** Filename - TapeTrap.bb2

AbsMouse 0,150
TapeTrap 0
RelMouse 380,0
TapeTrap
MouseWait
AbsMouse 0,150
PlayBack 0
VWait 30
End
```

QUIETTRAP

```
Mode(s):   Amiga
Statement: toggle TapeTrap execution mode
Syntax:    QuietTrap On/Off
```

This statement toggles the TAPETRAP execution mode. If QUIETTRAP is set to (On) then the BREXX commands recorded with TAPETRAP will not be displayed until they are played back. If QUIETTRAP is set to (Off) then the BREXX commands will be displayed as they are recorded. Here's an example:

```
; *** QuietTrap example
; *** Filename - QuietTrap.bb2

AbsMouse 0,150
; *** Display mouse movement
; *** when creating tape
QuietTrap Off
TapeTrap 0
RelMouse 380,0
TapeTrap
MouseWait
; *** Don't display mouse movement
; *** when creating tape
QuietTrap Off
AbsMouse 0,150
TapeTrap 0
RelMouse 0,200
TapeTrap
PlayBack 0
VWait 30
End
```

15.6.7 Macro keys

MACROKEY

```
Mode(s):   Amiga
Statement: define a macro key
Syntax:    MacroKey TAPE#,RAW_CODE,QUALIFIER
```

MACROKEY attaches a tape object to the keyboard, so that one key can be used to start the tape playing. The RAW_CODE and QUALIFER parameters are used to specify the macro key. Here are some examples:

```
; *** MacroKey example ** Filename - MacroKey.bb2

AbsMouse 0,150
TapeTrap 0
RelMouse 380,0
TapeTrap
; *** Press HELP key
MacroKey 0,95,0
MouseWait
End
```

```
; *** MacroKey example 2 ** Filename - MacroKey2.bb2

AbsMouse 150,0
TapeTrap 0
RelMouse 0,200
TapeTrap
; *** Press escape key
MacroKey 0,69,0
MouseWait
End
```

FREEMACROKEY

```
Mode(s):   Amiga
Statement: remove a macro key
Syntax:    FreeMacroKey RAW_CODE,QUALIFIER
```

This statement removes the tape object attached to the specified macro key. For example:

```
; *** FreeMacroKey example ** Filename - FreeMacroKey.bb2

AbsMouse 0,150
TapeTrap 0
RelMouse 380,0
TapeTrap
; *** Assign macro to HELP key
MacroKey 0,95,0
; *** Remove macro key
FreeMacroKey 95,0
MouseWait
End
```

15.7 End-of-Chapter summary

The NTSC and DISPHEIGHT functions should be used when designing software which can dynamically adjust itself to different displays (NTSC or PAL). DISPHEIGHT returns a value of 256 if it is executed in PAL mode, or 200 if executed on an NTSC Amiga.

Objects can be Blitz Basic screens, Slices, BitMaps, shapes etc. The USE, FREE, USED, ADDR and MAXIMUM commands are used to find out information about these objects.

WBHEIGHT, WBWIDTH, WBDEPTH and WBVIEWMODE are used to find out information about the Workbench screen.

Tape objects are predefined sequences of events which can be played back at any time, independently of the main program.

Chapter 16 : Advanced programming

The chapter contains information on conditional compilation techniques. It also covers the commands related to Blitz Basic's in-line assembler.

16.1 Compiler directives

The following section covers the commands which affect how a program is compiled by Blitz Basic.

16.1.1 Include files

These are files of predefined data which may be "included" in a source code file using a special directive. When the program is compiled these additional files are compiled as part of the main source code.

INCLUDE

```
Directive: compile a file as part of the current program
Syntax:    INCLUDE "FILENAME"
```

The INCLUDE directive is used to compile an external file as part of the current Blitz Basic program. The file must be a tokenised Blitz 2 program. For example:

```
; *** INCLUDE example
; *** Filename - INCLUDE.bb2

; *** Create the following program
; *** and save it to df0: as "BITMAP.bb2"

;BLITZ
;BitMap 0,320,256,3
;Slice 0,44,3
;Show 0

; *** Then enter the following listing
INCDIR "df0:"
INCLUDE "BITMAP.bb2"
For A=1 To 300
  Plot Rnd(320),Rnd(DispHeight),Rnd(5)+1
Next A
MouseWait
End
```

XINCLUDE

```
Mode(s):   N/A
Directive: compile a file as part of the current program
Syntax:    XINCLUDE "FILENAME"
```

The XINCLUDE (exclusive include) directive works identically to INCLUDE, however XINCLUDE files are only included once, regardless of the number of times the same filename is used. Here's an example:

```
; *** XINCLUDE example
; *** Filename - XINCLUDE.bb2

; *** Create the following program
; *** and save it to df0: as "SCREEN.bb2"
;Screen 0,3
;ScreensBitMap 0,0
;Cls 0

; *** Then enter the following listing
INCDIR "df0:"
XINCLUDE "SCREEN.bb2"
For B=1 To 300
  Circlef Rnd(320),Rnd(DispHeight),Rnd(20)+10,Rnd(5)+1
Next B
MouseWait
End
```

INCBIN

```
Mode(s):   N/A
Directive: compile a binary file as part of the current program
Syntax:    INCBIN "FILENAME"
```

The INCBIN directive is used to include a binary file as part of the current Blitz Basic program. For example:

```
; *** INCBIN example
; *** Filename - INCBIN.bb2

INCDIR "df0:"
INCBIN "Binary file"
MouseWait
End
```

INCDIR

```
Mode(s):   N/A
Directive: specify file path for INCLUDE directives
Syntax:    INCDIR PATH
```

INCDIR is used to specify the correct filename path for the INCLUDE, XINCLUDE and INCBIN directives. Example:

```
; *** INCDIR example
; *** Filename - INCDIR.bb2

INCDIR "RAM:"
INCLUDE "A FILE.bb2"
MouseWait
End
```

RUNERRSOFF

```
Mode(s):   N/A
Directive: disable error checking
Syntax:    Runerrsoff
```

The RUNERRSOFF directive can be used to disable error checking in different parts of your programs. It overrides the COMPILER OPTIONS settings. This allows you to make mistakes, without the Blitz Basic editor reporting them:

```
; *** Runerrsoff example
; *** Filename - Runerrsoff.bb2

Runerrsoff
; *** Should generate error
Blit 0,100,100
MouseWait
End
```

RUNERRSON

```
Mode(s):   N/A
Directive: enable error checking
Syntax:    Runerrson
```

The RUNERRSON directive enables error checking. It also overrides the COMPILER OPTIONS settings.

```
; *** Runerrson example
; *** Filename - Runerrson.bb2

Runerrsoff
; *** Should generate error
Blit 0,100,100
Runerrson
; *** Does generate error
Blit 0,100,100
MouseWait
End
```

16.1.2 Conditional compiling

Conditional compiling alows you to select which parts of a program are compiled by Blitz Basic. It can be used to produce "crippled" demonstration software (e.g. a utility with the save function disabled).

CNIF

```
Mode(s):   N/A
Directive: define start of conditional compiling
Syntax:    CNIF CONSTANT1 COMPARISON CONSTANT2
```

CEND

```
Mode(s):   N/A
Directive: end conditional compiling definition
Syntax:    CEND
```

The CNIF directive is used to set the start of a conditional compiling definition. It allows you to conditionally compile a section of code based on a comparison of two constants. The COMPARISON parameter should be one of the following Blitz Basic operators:

Table 16.1 : Conditional compiling operators

```
Operator Description              Example
===========================================
<        Less than                CNIF A<B
>        Greater than             CNIF A>B
=        Equal                    CNIF A=B
<>       Unequal                  CNIF A<>B
<=       Less than or equal to    CNIF A<=B
>=       Greater than or equal to CNIF A>=B
```

If the comparison is true then the code will be compiled, but if it is false then the program code will not be compiled until a CEND directive is executed.

CEND terminates a conditional compiling definition. Try the following example:

```
; *** CNIF example
; *** Filename - CNIF.bb2

#DISABLED=0
BLITZ
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
CNIF #DISABLED=1
  For B=1 To 50
    Locate Rnd(30),Rnd(20)
    Colour Rnd(5)+1
    NPrint "Blitz Basic"
  Next B
  MouseWait
CEND
End
```

In the above example the whole program is not compiled, as the #DISABLED constant is set to (0). However, if the #DISABLED constant is set to (1) then the whole program will be compiled.

CSIF

```
Mode(s):   N/A
Directive: compile based on string comparison
Syntax:    CSIF "STRING" COMPARISON "STRING"
```

The CSIF directive is used to set the start of a conditional compiling definition. It allows you to conditionally compile a section of code based on a comparison of two text strings. The COMPARISON parameter should be one of the previous Blitz Basic operators.

If the comparison is true then the code will be compiled, but if it is false then the program code will not be compiled until a CEND directive is executed.

CEND must be used to terminate the definition. Here is an example:

```
; *** CSIF example
; *** Filename - CSIF.bb2

BLITZ
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
CSIF "String"="String"
  NPrint "Compiled!"
CELSE
  NPrint "Not compiled!"
CEND
MouseWait
End
```

CELSE

```
Mode(s):   N/A
Directive: compile when a comparison is false
Syntax:    CELSE
```

The CELSE directive is used in conjunction with CNIF and CEND, or CSIF and CEND, to qualify a condition. The commands between CNIF or CSIF and CELSE are compiled when the logical condition following CNIF or CSIF is true.

If the condition following CNIF or CSIF is false then the commands after CELSE are compiled instead. Here's an example:

```
; *** CELSE example
; *** Filename - CELSE.bb2

#DISABLED=0
BLITZ
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
CNIF #DISABLED=1
  For B=1 To 20
    VWait 5
    Cls Rnd(7)
  Next B
CELSE
  NPrint "Colour flash not compiled!"
  MouseWait
CEND
End
```

CERR

```
Mode(s):   N/A
Directive: generate compile-time error message
Syntax:    CERR MESSAGE
```

The CERR directive is used to generate compile-time error messages. CERR is normally used in conjunction with macros and conditional compiling to generate errors when incorrect macro paramaters are passed. For example:

```
; *** CERR example
; *** Filename - CERR.bb2

Macro ERROR
  ; *** One parameter passed to macro?
  CNIF `0=1
    CERR "Correct number of parameters"
  CELSE
    ; *** Wrong number of parameters
    CERR "Illegal number of macro parameters"
  CEND
End Macro

!ERROR{A}
MouseWait
End
```

16.1.3 Macros

A macro is simply a list of commands which is called up by typing the macro name (preceded by an exclamation mark). Once a macro has been defined it can be used anywhere in the program, and will have exactly the same effect as if the assigned comamnds had been entered from the keyboard.

MACRO

```
Mode(s):   N/A
Directive: declare start of macro definition
Syntax:    Macro NAME
```

END MACRO

```
Mode(s):   N/A
Directive: end a macro definition
Syntax:    End Macro
```

The MACRO directive is used to declare the start of a macro definition. All commands following this directive are included in the macro's contents. END MACRO terminates a macro definition. Here are some examples:

```
; *** Macro example
; *** Filename - Macro.bb2

Macro AMACRO
  NPrint "This is a macro"
  NPrint "definition"
End Macro

; *** Call macro 10 times
For A=1 To 10
  !AMACRO
Next A
MouseWait
End
```

```
; *** Macro example 2
; *** Filename - Macro2.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0

Macro ANOTHERMACRO
  For A=1 To 200
    Plot Rnd(320),Rnd(256),Rnd(5)+1
  Next A
End Macro
```

```
; *** Call macro 10 times
For B=1 To 10
  Cls 0
  !ANOTHERMACRO
  VWait 25
Next B
End
```

16.2 Assembler

Assembly language can be included either in-line, using GETREG and PUTREG to access variables, or inside procedures. This allows the experienced Amiga programmer to speed up processor-intensive BASIC with faster machine code.

Blitz Basic's in-line assembler is very easy to use (if you are an experienced 68000 programmer!). All 68000 mnemonics are tokenised as if they were BASIC reserved keywords, and are assembled into machine code when your program is compiled. However, the in-line assembler does have a couple of limitations: the Absolute Short addressing mode and Short Branches are not supported, and assembler expressions must use curly brackets ({}) to force operator precedence.

The in-line assembler can be treated as an assembler instead of a compiler, although why you'd want to do this is not certain (surely Devpac 3.0, with its superior debugging tools is a much more stable assembler!?).

However, in order to correclty mix assembler with BASIC then the following rule must be obeyed:

Address registers A4-A6 must be preserved and restored by any assembly language routines. Blitz 2 uses A5 as a global variable base, A4 as a local variable base, and tries to keep A6 from having to be re-loaded too often.

DC

```
Mode(s):   Amiga/Blitz
Statement: define constant
Syntax:    Dc[.SIZE] DATA[,DATA...]
```

DC stands for Define Constant, and is used to define areas of data for assembly language programs.

DCB

```
Mode(s):   Amiga/Blitz
Statement: define constant block
Syntax:    Dcb[.SIZE] REPEATS,DATA
```

DCB stands for Define Constant Block. It is used to insert a repeating series of the same value into assembly language programs.

DS

```
Mode(s):   Amiga/Blitz
Statement: define storage
Syntax:    Ds[.SIZE] LENGTH
```

DS stands for Define Storage. It is used to insert a gap into a program, which may be used as a data storage area.

EVEN

```
Mode(s):   Amiga/Blitz
Statement: word align internal program counter
Syntax:    Even
```

EVEN is used to word align the Blitz Basic internal program counter. This may be necessary if a DC, DCB or DS statement has left the program counter at an odd address.

GETREG

```
Mode(s):   Amiga/Blitz
Statement: transfer result to register
Syntax:    GetReg 68000 REG,EXPRESSION
```

GETREG is used to transfer the result of a BASIC expression to a 68000 register. The result of the expression is converted into a long value before being moved to the data register.

GETREG should only be used to transfer expressions to one of the eight data registers (d0-d7). GETREG uses the stack to temporarily store any registers used in the calculation of the expression. See PUTREG examples.

PUTREG

```
Mode(s):   Amiga/Blitz
Statement: transfer register to variable
Syntax:    PutReg 68000 REG,VARIABLE
```

The PUTREG statement is used to transfer a value from any 68000 register (d0-d7/a0-a7) into a variable. If the specified variable is a string, long, float or quick, then all four bytes from the register will be transferred. If the specified variable is a word or a byte, then only the relevant low bytes will be transferred. Here are some examples:

```
; *** GetReg/PutReg example
; *** Filename - PutReg.bb2

A.w=100
; *** Put variable into register
GetReg d0,A
; *** Negate number
NEG d0
; *** Grab contents of register
PutReg d0,B.w
NPrint B
MouseWait
End
```

```
; *** GetReg/PutReg example 2
; *** Filename - PutReg2.bb2

A.w=100
B.w=5
; *** Put variables into registers
GetReg d0,A
GetReg d1,B
; *** Add register 0 to register 1
ADD d0,d1
; *** Grab contents of register 1
PutReg d1,C.w
NPrint C
MouseWait
End
```

SYSJSR

```
Mode(s):   Amiga/Blitz
Statement: call system routine
Syntax:    SysJsr ROUTINE
```

SYSJSR is used to call Blitz Basic's system routines. The ROUTINE parameter specifies the routine number to call.

TOKEJSR

```
Mode(s):   Amiga/Blitz
Statement: call library based routine
Syntax:    TokeJsr TOKEN[,FORM]
```

TOKEJSR is used to call Blitz Basic's library-based routines. The TOKEN parameter specifies a token number, or token name. FORM is a particular form of the token.

ALIBJSR

```
Mode(s):   Amiga/Blitz
Statement: call routine from one library to another
Syntax:    ALibJsr TOKEN[,FORM]
```

ALIBJSR is used to call a routine from one library to another. Please refer to the library writing section of the programmers guide for more information on library writing.

BLIBJSR

```
Mode(s):   Amiga/Blitz
Statement: call routine from one library to another
Syntax:    BLibJsr TOKEN[,FORM]
```

BLIBJSR is used to call a routine from one library to another. Please refer to the library writing section of the programmers guide for more information on library writing.

16.3 End-of-Chapter summary

Includes are files of predefined data which may be "included" in a source code file using a special directive. When the program is compiled these additional files are compiled as part of the main source code.

Conditional compiling alows you to select which parts of a program are compiled by Blitz Basic. It can be used to produce "crippled" demonstration software (e.g. a utility with the save function disabled).

A macro is simply a list of commands, which is called up by typing the macro name. Once a macro has been defined it can be used anywhere in the program, and will have exactly the same effect as if the assigned comamnds had been entered from the keyboard.

Blitz Basic incorporates a full in-line Assembler. Address registers A4-A6 must be preserved and restored by any assembly language routines. Note that the Absolute Short addressing mode and Short Branches are not supported, and assembler expressions must use curly brackets ({}) to force operator precedence.

Chapter 17 : Program startup

This chapter covers all of the commands involved in program startup. It will teach you all you need to know about executable files, parameter passing and Blitz Basic runtime.

17.1 Executable files

Executable files are those which can be run independently from the Blitz Basic 2 environment.

Creating an executable file is easy. First, load the program you wish to create an executable file for into the Blitz Basic editor. If you want Blitz to create an icon for your program then make sure that the "Create Icons for Executable Files" option in the COMPILER OPTIONS is set to on (i.e. highlighted). Then select "CREATE EXECUTABLE" from the COMPILER menu, or press Amiga & E. Type in the filename of the program you wish to create and press the return key. Blitz Basic will then create your executable file.

In order for any Blitz Basic program to run from outside of the Blitz Basic 2 system, the following statement MUST be placed at the very beginning of your program code. If it is not included then your program will not run from Workbench, it's as simple as that.

WBSTARTUP

```
Mode(s):   Amiga/Blitz
Statement: allow program to run from Workbench
Syntax:    WBStartup
```

If you want your Blitz Basic creation to run from Workbench then you MUST include this statement in the program, otherwise, the program will crash. You have been warned! Here are some examples:

```
; *** WBStartup example
; *** Filename - WBStartup.bb2

WBStartup
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
Locate 0,3
NPrint "I will start from Workbench"
MouseWait
End
```

```
; *** WBStartup example 2
; *** Filename - WBStartup2.bb2

; *** No WBSTARTUP statement!!!
BLITZ
```

```
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
Locate 0,3
NPrint "I won't start from Workbench"
MouseWait
End
```

CLOSEWORKBENCH_

```
Mode(s):   Amiga/Blitz
Statement: close Workbench
Syntax:    CloseWorkBench_
```

This statement closes the Workbench screen, freeing about 40K of valuable memory for your own programs. Example:

```
; *** CloseWorkBench_ example
; *** Filename - CloseWorkBench_.bb2

CloseWorkBench_
MouseWait
End
```

OPENWORKBENCH_

```
Mode(s):   Amiga/Blitz
Statement: open Workbench
Syntax:    OpenWorkBench_
```

This statement reopens the Workbench screen. For example:

```
; *** OpenWorkBench_ example
; *** Filename - OpenWorkBench_.bb2

OpenWorkBench_
MouseWait
End
```

17.1.1 CLI Parameters

When a program is run from the CLI, various parameters can be passed to the program. This allows the user to alter, or influence, various aspects of the program without getting their hands dirty in the actual program code.

NUMPARS and PAR$ return the number of parameters and grab the contents of the CLI command line respectively. They can be used to return strings containing all the parameters which have been entered from the command line. If parameters have not been defined, you'll get (0) from NUMPARS, or a null string ("") from PAR$ instead.

For example, if you wanted to run an executable file from the CLI and tell it to open a high-resolution screen, then you would call the program something like this:

```
Program HIGHRES
```

The following code could be used to read this parameter and set the screen mode accordingly:

```
; *** Passing parameters
; *** Filename - Pass.bb2

If Par$(1)="HIGHRES"
  Screen 0,11,"High-res"
Else
  Screen 0,3,"Low-res"
EndIf
MouseWait
End
```

Another interesting use of parameter passing is illustrated in the following example. If you wanted to tailor software to the user, without having to prompt them for questions about name, age etc., then you could get them to call your program with the parameters as their details:

```
Program NEIL,WRIGHT,197
```

Your program could then display this information upon loading:

```
; *** Parameter passing 2
; *** Filename - Pass2.bb2

BLITZ
BitMap 0,320,256,3
BitMapOutput 0
Slice 0,44,3
Show 0
```

```
If NumPars=3
  Colour 1
  ; *** First parameter
  NPrint "First Name: ",Par$(1)
  Colour 2
  ; *** Second parameter
  NPrint "Surname: ",Par$(2)
  Colour 3
  ; *** Third parameter
  NPrint "Age: ",Par$(3)
End If
MouseWait
End
```

The "CLI Arguement" menu option can be used to test CLI parameters. Here, parameters can be entered and tested on non-executable programs prior to the creation of executable files. These parameters may be read by the NUMPARS and PAR$ functions.

NUMPARS

```
Mode(s):  Amiga/Blitz
Function: return number of parameters passed to program
Syntax:   p=NumPars
```

The NUMPARS function returns the number of parameters passed to an executable program from either the Workbench or the CLI. If no parameters are supplied then NUMPARS will return (0).

For example, if a Blitz Basic program is called in the following way, through the CLI, then (2) is returned because two parameters are passed:

```
Program PARAMETER1 PARAMETER2
```

Programs run from Workbench, however, are only capable of picking up one parameter. This is achieved through the file's "Default tool" (contained in the .Info file), or by multiple selection through the shift key. Here's an example:

```
; *** NumPars example
; *** Filename - NumPars.bb2

NPrint NumPars," parameters passed"
MouseWait
End
```

PAR$

```
Mode(s):  Amiga/Blitz
Function: return string equivalent to a passed parameter
Syntax:   p$=Par$(PARAMETER)
```

The PAR$ function returns a string equivalent to a parameter passed to an executable file through either the CLI or Workbench. If no parameters have been defined, a null string ("") is returned.

Example:

```
BLASTER one two three
```

Runs a program called "BLASTER", with parameters "one", "two" and "three". These parameters can be returned by the following code:

```
; *** Par$ example
; *** Filename - Par$.bb2

For A=1 To NumPars
  NPrint Par$(A)
Next A
MouseWait
End
```

17.2 Runtime program startup

The following commands have no effect on programs run outside of the Blitz Basic 2 environment (i.e. programs that have been converted to executable files). They are primarily of use during program development.

CLOSEED

```
Mode(s):   Amiga/Blitz
Statement: close Blitz Basic editor
Syntax:    CloseEd
```

This statement is used to close the Blitz Basic 2 editor screen when programs are executed from within Blitz Basic. This frees around 40K of valuable chip memory. CLOSEED has no effect on executable files run outside of the Blitz environment. Here's an example:

```
; *** CloseEd example
; *** Filename - CloseEd.bb2

CloseEd
NPrint "No editor!!!"
MouseWait
End
```

NOCLI

```
Mode(s):   Amiga/Blitz
Statement: prevent default CLI from opening under Blitz Basic
Syntax:    NoCli
```

NOCLI prevents the default CLI from opening when programs are executed from within Blitz Basic. This has no effect on executable files run outside of the Blitz environment. For example:

```
; *** NoCli example
; *** Filename - NoCli.bb2

NoCli
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
Locate 0,3
NPrint "No default CLI"
MouseWait
End
```

17.3 End-of-Chapter summary

If an executable file is to start from Workbench then the WBSTARTUP statement must be the very first line in your program.

The NUMPARS function returns the number of parameters passed to an executable file. PAR$ returns a string equivalent to a parameter passed to an executable file through either the CLI or Workbench.

CLOSEED can be used to close the Blitz Basic 2 editor. This frees around 40K of valuable chip memory.

Executing the NOCLI statement prevents the default CLI from opening.

Chapter 18: The Future

For the duration of this guide's 400 pages I have tried to help you get the most from Blitz Basic 2 by providing a comprehensive command reference. By this stage you should have a reasonable knowledge of the Blitz instruction set and of the techniques that can be used to create professional software in BASIC. I hope you have enjoyed the journey!

If you want to hone your coding skills and make contact with other Blitz Basic programmers then join one of the many user clubs. They are all run by friendly, helpful folk whose main goal in life is to make your coding a little less painful. For really up to date information the only way to turn is to the Blitz User Group.

18.1 The Blitz User Group

B.U.G is a new (at the time of writing) club dedicated to all aspects of Blitz Basic 2. The aims of the club are simple: to teach people how to create commercial quality games in Blitz and make as much money from their programs as possible. Or something.

Each issue of the bi-monthly B.U.G disk magazine contains all the latest Blitz news, reviews, tips and previews, together with comprehensive tutorials, loads of Blitz source code and plenty of free commercial quality graphics.

Each issue costs £3.50 and is available from:

```
Seasoft Computing
Unit 3
Martello Enterprise Centre
Courtwick Lane
Littlehampton
West Sussex
BN17 7PA
Tel: (01903) 850378
```

18.2 Blitz User International

The Blitz User International Magazine is produced by Matthew Tillett & Kevin Winspear, in the UK, to provide support for all Blitz programmers. B.U.I is a monthly printed magazine, with coverdisk, which covers the main aspects of Blitz programming. If you would like to join this group, or purchase their fanzine, then write to (with S.A.E):

```
B.U.I
27 Hillside Avenue
Worlingham
Beccles
Suffolk
NR34 7AJ
```

18.3 Blitz User Magazine

The unfortunately named BUM magazine is the official offering from Acid Software, the producers of Blitz Basic. Your £15 subscription fee gets you five issues full of extensions, upgrades, source code and other goodies. All users who register with Acid Software in the United Kingdom will receive information on the Blitz User Magazine:

```
Acid Software Distribution Centre
Unit 15 Guildhall Industrial Estate
Kirk Sandle
Doncaster
England
DN3 1QR
Tel: (01302) 890000
```

18.4 Blitz User Disk Magazine

If you want to communicate with Blitz users nationwide, here is your chance. The Blitz User Disk Magazine is produced by Michael Milne, in the UK, to provide local support for all Blitz Basic users. It covers the main features of Blitz Basic and has sections on programming, graphics, music and new Amiga hardware.

The B.U.D (great acronym, guys) magazine covers the main areas of interest to Blitz users with routines as its main feature. Users can talk about themselves, their interests, equipment and exchange information with fellow programmers.

Readers are encouraged to submit programs for publication and Michael will try his best to answer any programming queries. If you would like to participate in this user club you can obtain a free copy of the first issue by sending a blank disk and SAE to the following address:

```
Blitz User Disk Magazine
c/o 39 The Drive
Earley
Berkshire
England
RG6 1EG
```

18.5 Magazine columns

As well as the various Blitz clubs and societies, there are a number of Blitz-dedicated columns in the monthly Amiga magazines. **Amiga Format** and **Amiga Shopper** are both good sources of Blitz Basic information. You may also be interested in **CU Amiga** semi-regular column which aims to teach the fabled art of games programming, often with varying degrees of success.

18.6 Useful contacts

Acid Software headquarters, New Zealand. To contact the authors of Blitz Basic 2, write to this address:

```
Acid Software
10 St Kevins Arcade
Auckland 1
New Zealand
Tel: (64)-9-358 1658
```

Gothik PD specialise in Blitz Basic 2 programming and public domain software:

```
Gothik PD
7 Denmark Road
Northampton
England
NN1 5QR
Tel: (01604) 22456
```

For a comprehensive selection of Amiga Public Domain, including Blitz Basic games, demos and utilities, write to the following address. For £1 you will recive a catalogue listing over 8000+ titles:-

```
Tornado Software
10 Colenso Street
Hartlepool
Cleveland
TS26 9BD
```

(Cheques & PO's Payable to K. Winspear)

Appendix A : Blitz Basic Applications

There are a clutch of useful utilities available for the Blitz Basic 2 development system. Obviously we have Shapesmaker and MapEdit, the two utilities bundled with Blitz Basic, but we also have Shape-Ed V2 and BobEd, two superior object editors, and Blitz-Case, a complete software engineering package. Naturally, all of these applications were written using the Blitz Basic language.

One of the many pitfalls of the Blitz Basic 2 documentation is that it does not contain details of the Shapesmaker and MapEdit programs. Here are those illustrious instructions!

A.1 Shapesmaker

The Amiga range of computers have access to an extremely powerful graphic shifter called the Blitter chip. Blitter Objects, or "Bobs" for short, are images which can be displayed on screen with lightning speed, but must be displayed and updated by the user to avoid graphic corruption. For reasons know only to Acid Software, Blitz Basic refers to these Bobs as shapes, or shape objects. These shape objects may be used in a variety of different ways, such as gadgets, menu items or game graphics.

The Shapesmaker utility can be found in the "tools" directory of the Blitz Basic 2 System disk. This program was, until a short while ago, the only way of producing shape objects for your Blitz programs.

When designing your shapes in a utility such as Deluxe Paint IV, you should make sure that each row of objects is aligned on exactly the same line across the screen, otherwise they will not be recognised by Blitz Basic. Also, all shapes on the same row must be of the same vertical size (the horizontal size is not important). This is crucial if you want your shapes to be recognised by the shape editor.

After the Shapesmaker program has loaded, hold down the right-hand mouse button. You should be greeted by the following list of menu options:

```
PROJECT  OPTIONS      COLOUR
------------------------------------
LOAD     MAKE SHAPES  REMOVE COLOUR?
CONVERT  MAKE SPRITES
QUIT     AUTO CENTRE
```

Shapesmaker is fully menu-driven so I will run through each of the menu options in turn and (hopefully) give you a complete guide to shape creation.

A.1.1 Loading an IFF file

To load an IFF file into memory you select the LOAD menu option (located under the PROJECT menu) and input the relevant filename.

The QUIT option is also located under the PROJECT menu, but you won't want to select that just yet.

A.1.2 Shapes or sprites?

The next step is to tell Shapesmaker which type of object you will be dealing with, either shapes or sprites. If you look under the OPTIONS menu then you will see that the MAKE SHAPES option is selected. So, if you want to create shape objects then you can leave this option alone, but if you want to create some sprites then select MAKE SPRITES.

A.1.3 Detention centre

The AUTO CENTRE option is a handy little function which tells Shapesmaker to automatically centre each object's hot spot, or handle, as they are converted. An object's handle refers to an offset, in pixels, from the upper left of the shape to be used when calculating a shape's coordinates. Consult Chapter 8 for further information.

A.1.4 Masking tape

When designing shapes and sprites, it often helps to set aside a single colour which can be used as a background "mask" colour. To create a mask, simply create a series of filled rectangle of size greater than your largest shape, then paste each shape on top of a rectangle. When you convert your IFF files to shape objects using the editor, you can automatically remove this background mask by selecting the REMOVE COLOUR? option from the COLOUR menu.

Using a background mask allows you to overcome the vertical sizing restriction imposed by the shape editor. Say, for example, that you had drawn an animating alien that increased in size from frame to frame. Under normal circumstances Shapesmaker wouldn't recognise the shapes, but if a background mask was used then all of the objects would be of the same vertical size.

A.1.5 Preaching to the converted

Having loaded your IFF file and set the object type, handle offset, and background mask, the final step is to convert the picture to a series of shape objects. This is achieved by selecting CONVERT from the PROJECT menu. Naturally you will be asked to input a filename and, all being well, Shapesmaker will chop up your graphics and save them onto a disk.

Whilst Shapesmaker lacks the raw power of utilities such as BobEd and Shape-Ed V2, it more than makes up for it in its ease of use.

A.2 MapEdit

The MapEdit program can also be found in the "tools" directory of the Blitz Basic 2 System disk. MapEdit is a fabulous utility which enables you to create the backgrounds used in mapped games. It allows you to create giant screens in memory using a series of simple building blocks, much like a jigsaw or collage. You can design giant maps, the height of many screens and they will not take up nearly as much memory as normal pictures.

The practical use of MapEdit can be seen in many different commercial games, such as Woody's World (a cute-style platform game) and Roadkill (an overhead driving game written entirely in Blitz Basic 2!).

A.2.1 First steps

First you must create your blocks using a paint package such as Deluxe Paint IV. Blocks are simple square or rectangular areas that can be glued together in varying arrangements and used again and again on the same background. The actual size of the blocks is up to you, but common sizes are 16*16 pixels, 32*32 pixels, 32*16 pixels and 16*32 pixels. Bear in mind that all blocks must be of the same size (i.e. all 16*16 pixels, or all 32*32 pixels) otherwise they will be of no use to MapEdit.

The next step is to use the Shapesmaker utility to grab your screen blocks from an IFF file and convert them to shape objects (see above for more details). Then you can use MapEdit to paste these building blocks together, thus creating a complete game backdrop.

A.2.2 Starting from scratch

Upon loading the MapEdit utility you will be greeted by a list of parameters, much like the one below:

```
Map Parameters
--------------
Block Width:    16
Block Height:   16
Map Width:      20
Map Height:     12
```

The Block Width and Block Height parameters should be set to the same dimensions as your graphic blocks. The Map Width and Map Height parameters are the width and height of your map respectively, measured in screen blocks.

For example, if your blocks are 16*16 pixels in size and your map width is 20 blocks then the total width is (16*20) = 320 pixels. Similarly, if 32*16 pixel blocks are used and the map width is set to 20 then the total map width will be (32*20) = 640 pixels, or two screens in width!

Let's leave these values as they are for the moment, so click once on the USE gadget and we will begin our journey. MapEdit is also fully menu-driven; here are the various menu options:

```
PROJECT  BLOCKS        GROUP
----------------------------
NEW      LOAD SHAPES   CREATE
LOAD     LOAD PALETTE  DELETE
SAVE
QUIT
```

The NEW option is primarily of use when you want to clear the map editing screen. This also erases any maps from memory, so remember to save your work before you access this function!

The lOAD option loads a previously created map into memory. Try loading the "demomap" file from the "mapedit.demo" directory of your Blitz Basic 2 System disk. It doesn't look very exciting, does it? That's because we haven't loaded any screen blocks into memory. To do this, highlight the LOAD SHAPES option and load the "blocks.shapes" file, also from the "mapedit.demo" directory. Right, that's the map and blocks in place, but what's wrong with the screen colours? You've guessed it, we need to load the shapes' palette as well. To do this, highlight the LOAD PALETTE option and load the "blocks.palette" file, again from the "mapedit.demo" directory.

Not all of the blocks can be displayed at once. To cycle through all of the blocks, use the two white arrows at the top left and top middle of the screen. To paste a block on the map, simply click on it, move the mouse pointer to the relevant position and press the left mouse button. Easy when you know how!

A.2.3 I'm a map - edit me!

One of the options that we haven't covered so far is group creation. Groups are what Blitz Basic calls a number of blocks joined together. Say, for example, that you wanted to move a large section of a map about. Instead of moving it block by block, you would create a group and manipulate a large number of blocks around at once.

To create a group of screen blocks you have to highlight the CREATE option from the GROUP menu. This causes a small window, entitled "Making group - close me when done..." to appear. Now, highlight the blocks you want to join together, one by one, in order and then close the window by clicking on its close gadget. All being well your group should appear under the mouse pointer. Try moving it about - you can now manipulate this group as you would a single screen block.

As with single blocks, you will often find that there are too many groups to display at the top-right of the screen. To cycle through them use the white arrows at the top-middle and top-right of the display. To erase a group from memory, simply click on the group icon (at the top-right of the screen) and select the DELETE option from the GROUP menu.

A.3 Shape-Ed V2

Although the supplied Blitz Basic shape editor is adequate for most users needs, it does not allow direct manipulation of the physical appearance of shape objects. Shape-Ed V2, being a modified version of the original shape editor, comes complete with a fairly comprehensive array of tools designed specifically with this task in mind.

The editor is fairly easy to use and is an excellent example of how Blitz Basic can be used to create powerful Intuition-based applications.

Important features include:

- Convert AMOS Bobs to Blitz shapes
- Rotate and mirror shapes
- Fill, airbrush, text
- Draw circles, rectangles, lines
- Cut and paste shapes
- Comprehensive animation displayer
- Magnify shapes

Blitz users who register with Eclipse Software receive the full program, together with the fully annotated source code and a laser printed manual. To purchase Shape-Ed V2 send a cheque for seven pounds, made payable to Aaron Sethi, to the following address:

```
Eclipse Software
144 Ravenbourne Avenue
Shortlands
Bromley
Kent
England
BR2 OAY
Tel: (081) 464 8416
```

A.4 BobEd V1.2

BobEd has been developed by Paul Thompson to "offer Blitz Basic users flexibility, power & convenience in the handling of shape objects". What sets BobEd above Shape-Ed and similar utilities is its comprehensive set of editing tools. Although the system has a feature list as long as your arm, some of the more important features are:

- 256 colour hi-res shape editing (AGA)
- Shape grabbing tools
- Palette manipulation
- Wide range of drawing tools and functions
- Handles brushes, anim-brushes, shapes and sprites

As well as an extremely powerful shape editor, the BobEd package includes a number of fully-annotated example programs which make use of BobEd-created shapes.

BobEd V1.2 costs just £11.95 inc P&P and is available from the following address:

```
Aspire 2
Strathspey
Pentre Hill
Flint Mountain
Clwyd
England
CH6 5QN
Tel: 0352 761798
```

A.5 Blitz-Case

Blitz-Case gives Blitz Basic programmers a CASE (Computer Aided Software Engineering) environment in which to code, allowing the programmer to mix easy flowchart symbols with source code.

The basic structure of a Blitz Basic program can be quickly designed and constructed using easy to understand flowchart icons. The exact source code is then assigned to certain icons before the flowchart is converted into Blitz Basic source code. There are icons to replace all the standard BASIC constructs such as:

- If...Then
- Select...End Select
- Procedures

Icons which can contain source code are called processes. There are three kinds of process icons in Blitz-Case. All perform exactly the same function - holding a set of BASIC commands - but the different types can be used to make comprehension of the flowchart easier. The PROCESS is the generic icon, the INPUT type is usually used for processes which handle user input, and PREPARATION icons are used for processes which essentially prepare for something else to happen. Anyone who can program Blitz Basic is the normal way will quickly pick up the CASE techniques.

At its simplest level, Blitz-Case can be used to plan out a program - outlining the procedures and processes that are likely to be required. At its most complex Blitz-Case can reduce the need for the Blitz Basic Compiler to simply final compilation, with all the editing and programming done in Blitz-Case.

The program is perhaps the biggest accessory yet to appear for Blitz Basic, and is only the second of its type to appear on the Amiga (Fed-Case being the other, which is for the C programming language). Full AmigaGuide instructions are included on the disk, although the program is fairly intuitive to use. If you are interested in Blitz-Case then contact the author, Richard Jones, at the following address:

```
Richard Jones
14 Torrington Avenue
Weeping Cross
Stafford
England
ST17 0HZ
```

Blitz-Case is just one of those things you have to have if you're serious about Blitz Basic. As you can probably tell, I was impressed!

Appendix B : Useful Programs

One of the best ways of learning any computer language is by typing in programs and routines and learning from the work of others. This appendix contains several handy programs and routines that can easily be adapted and incorporated into your own Blitz Basic creations. Thanks must go to all of the Blitz users who sent in their routines for inclusion. Where possible I have included programs that illustrate various programming techniques and solve some common programming problems. And you don't even have to type in the programs - they can all be found on disk 2 of the guide.

B.1 X-Plane starfield

One of the fiendishly clever, yet surprisingly simple routines is the parallaxing dot starfield. Parallaxing is a technique whereby speed is used to create an illusion of depth, and is achieved by moving individual pixels around at different speeds. This little proglet can throw up to 44 pixels about with no slow-down whatsoever, such is the speed of Blitz Basic. A few words of warning - because the routine draws up to 44 pixels on the screen in one frame, you really do need to turn the Runtime Error Debugger off for silky-smooth movement. Secondly, although the routine can display more than 44 pixels on screen, it will slow down to snail's pace.

You can customise the routine in a number of ways. To alter the number and speed of the stars simply change the values of the NUM and SPEED variables respectively. Try improving the routine so that stars moving at different speeds are plotted in different colours:

```
; *** X-Plane Starfield
; *** Filename - X-Plane_Starfield.bb2
; *** Author - Neil Wright

NEWTYPE .p
  X.w
  Y.w
  SPEED.w
End NEWTYPE

; *** Number of stars and maximum speed
NUM=40
MXSPEED=8

Dim List STAR.p(NUM)

BLITZ
; *** Open two BitMaps for double-buffering
BitMap 0,320,DispHeight,2
BitMap 1,320,DispHeight,2
Slice 0,44,320,DispHeight,$fff8,2,8,2,320,320
RGB 1,15,15,15
USEPATH STAR()
ResetList STAR()

; *** Generate random numbers
```

```
While AddItem(STAR())
  \X=Rnd(320)
  \Y=Rnd(DispHeight)
  \SPEED=Rnd(MXSPEED)+1
Wend

; *** Main loop
Repeat
  VWAIT
  Show MAP : MAP=1-MAP : Use BitMap MAP
  Cls
  Gosub STARS
Until Joyb(0)>0
End

; *** Star drawing sub-routine
STARS:
  USEPATH STAR()
  ResetList STAR()
  While NextItem(STAR())
    Plot \X,\Y,1
    \X=QWRAP(\X+\SPEED,0,320)
  Wend
Return
```

B.2 Z-Plane starfield

This little snippet of code creates a fantastic "hyperspace" effect by moving pixels outwards, from the centre of the screen, on the z-axis. Up to 115 stars can move in one frame (50Hz) with no slowdown, but remember to turn the Runtime Error Debugger off for maximum speed.

Again, you can customise the routine to your own needs. The number and speed of the stars can be controlled by altering the STARS and SPEED variables.

You may like to add a copper list to the background, or if you are feeling really adventurous, then try making the starfield rotate (hint: you may have to use SIN and COS to achieve this):

```
; *** Parallaxing Z-Plane Starfield
; *** Filename - Z-Plane_Starfield.bb2
; *** Author - Neil Wright

BLITZ
; *** Open two BitMaps for double-buffering
BitMap 0,320,DispHeight,1
BitMap 1,320,DispHeight,1
Slice 0,44,320,DispHeight,$fff8,1,8,2,320,320
RGB 1,15,15,15
Show 0

; *** Number of stars and speed
STARS=115
```

```
SPEED=1.07

Dim X(STARS),Y(STARS)

; *** Generate random numbers
For A=0 To STARS-1
  X(A)=Rnd(320)-160
  Y(A)=Rnd(DispHeight)-(DispHeight/2)
Next A

; *** Main loop
Repeat
  Cls 0
  For A=0 To STARS-1
    Plot X(A)+150,Y(A)+120,1
    X(A)=X(A)*SPEED
    Y(A)=Y(A)*SPEED
    If X(A)>160 Then X(A)=X(A)-160
    If Y(A)>160 Then Y(A)=Y(A)-160
    If X(A)<-160 Then X(A)=X(A)+160
    If Y(A)<-160 Then Y(A)=Y(A)+160
  Next A
  VWait
  Show MAP : MAP=1-MAP : Use BitMap MAP
Until Joyb(0)=1
```

B.3 Mandelbrot

Mandelbrots were first discovered by the IBM scientist Benoit Mandelbrot in the late 1970s. With Blitz Basic, fractals and Mandelbrots can be generated in a matter of minutes. You needn't be able to understand fractals in order to generate amazing pictures - all you need is the following listing:

```
; *** Simple Mandelbrot
; *** Filename - Mandelbrot.bb2
; *** Author - Neil Wright

BLITZ
BitMap 0,320,DispHeight,4
Slice 0,44,320,DispHeight,$fff8,4,8,16,320,320
Show 0

; *** Set up palette
For A=0 To 15
  RGB A,A,A,0
Next A

; *** Main loop
For Y=0 To 255
  For X=0 To 319
    X1=-2.0+X/81.92
    Y1=1.6-Y/81.92
    COL=0
    A=0
    C=0
    Repeat
      B=A*A-C*C+X1
      C=2*A*C+Y1
      A=B
      COL+1
    Until A*A+C*C>4 OR COL=16
    ; *** Plot pixel
    Plot X,Y,COL
  Next X
Next Y
MouseWait
End
```

B.4 Mirrored text

The Amiga's Copper (co-processor) chip is not only used to create colourful rainbow backgrounds. As the Copper executes its instructions in parallel with the main processor, fantastic hardware effects can also be generated. With careful use of the CUSTOMCOP statement we can mirror BitMap output on the y-axis:

```
; *** Mirrored text
; *** Filename - Mirror.bb2
; *** Author - Neil Wright

BLITZ
; *** A touch of hardware trickery
#BPLMOD1=$108
#BPLMOD2=$10A
BitMap 0,320,400,3
Line 0,200,320,200,5
BitMapOutput 0
Colour 4

; *** Text to mirror
A$="Blitz BASIC is tops!"
Locate 40/2-(Len(A$)/2),30
NPRINT A$
A$="(C) Blitz User Group"
Locate 40/2-(Len(A$)/2),32
NPrint A$
Slice 0,44,320,220,$fff8,3,8,32,320,320
; *** More hardware tom-foolery
CO$=Mki$(#BPLMOD1)+Mki$(-122)
CO$+Mki$(#BPLMOD2)+Mki$(-122)
CustomCop CO$,150+44
For A=1 To 180
```

```
  VWait
  Show 0,0,A
Next
MouseWait
End
```

B.5 System reset

For all the tech-heads, here is a useful routine written by Noel Baldacchino which resets the computer (much the same as pressing the [CTRL] and [Amiga] keys together). It is an excellent example of the power of Blitz Basic's in-line assembler. Remember to save your work before you execute the program:

```
; *** System reset routine
; *** Filename - Reset.bb2
; *** Author - Noel Baldacchino

SysReset:  MOVE.l   $4,a6          ; *** ExecBase
           JSR      -150(a6)       ; *** SuperState()
           LEA      $fc0002,a0
           MOVE.l   a0,$20
           JSR      (a0)           ; *** Go for it!
```

B.6 DF1: test

Another routine by Noel Baldacchino, this useful snippet of code is used to find out if DF1: (an external disk drive) is present. Use it when designing multi-disk games to find out if another drive is available, in order to minimise disk swapping:

```
; *** DF1: test routine
; *** Filename - Test.bb2
; *** Author - Noel Baldacchino

Gosub DF1_test

If DF1
  NPrint "DF1: is available."
  Else NPrint "No DF1!"
EndIf
MouseWait
End

.DF1_test
      MOVE.l $4,a6
      MOVEQ  #0,d0
      LEA    ResourceName(pc),a1
      JSR    -498(a6)
      MOVE.l d0,a0
      TST.l  52(a0)
      BEQ    DF1_OK:
only_DF0:
      DF1=False
      Return
DF1_OK:
      DF1=True
      Return

ResourceName: Dc.b "disk.resource",0
```

B.7 Splerge!

If you know your AMOS then you will be familiar with the Splerge! routine. Basically it stretches an image off the screen, starting at the bottom. Why not improve the routine so that the image stretches onto the screen, or fades away as it disappears?:

```
; *** Splerge using the Copper
; *** Filename - Splerge.bb2
; *** Author - Neil Wright

#BPLMOD1=$108
#BPLMOD2=$10A
BitMap 0,320,DispHeight,5
; *** Load an IFF screen to splerge
LoadBitMap 0,"INSERT YOUR OWN FILENAME HERE",0
VWait 100
BLITZ
Slice 0,44,320,DispHeight,$fff8,5,8,32,320,320
; *** Hardware bashing
CO$=Mki$(#BPLMOD1)+Mki$(-41)
CO$+Mki$(#BPLMOD2)+Mki$(-41)
Show 0
Use Palette 0
VWait 50
CustomCop CO$,DispHeight
; *** Stretch the display
For A=DispHeight To 0 Step -1
  CustomCop CO$,A
  VWait
Next
VWait
End
```

B.8 Fireworks

This program, based on an original routine by Paul Thompson, throws a firework into the air and explodes it. Try adding a splash of colour to the explosion (hint: use a copper list in the background) and perhaps a few more fireworks here and there. Remember to turn the Runtime Error Debugger off for maximum speed:

```
; *** Fireworks
; *** Filename - Fireworks.bb2
; *** Author - Paul Thompson

BLITZ
; *** Initialise arrays
Dim X(100),Y(100),XS(100),YS(100)
BitMap 0,320,256,1
BitMap 1,320,256,1
Slice 0,44,1
Use BitMap 0
RGB 0,0,0,0
RGB 1,15,15,15

Repeat
  TIM=0 : X=160 : Y=255
  XS=Rnd(4)-2 : YS=-Int(Rnd(3)+9)
  Repeat
    ; *** Launch rocket
    Cls
    Boxf X,Y,X+1,Y+1,1
    Y+YS
    YS+0.25
    X+XS
    VWait
    ; *** Double-buffer
    Show MAP : MAP=1-MAP : Use BitMap MAP
  Until YS=0
  For A=0 To 90
    X(A)=X
    Y(A)=Y
    XS(A)=Int(Rnd(15)-7)
    YS(A)=Int(Rnd(15)-7)
  Next A
  Repeat
  ; *** Explode!
    For B=0 To 30
      Plot X(B),Y(B),1
      X(B)+XS(B)
      Y(B)+YS(B)
      YS(B)+.25
    Next B
    VWait
    ; *** Double-buffer
    Show MAP : MAP=1-MAP : Use BitMap MAP
    Cls
    TIM+1
  Until TIM=>100
Until Joyb(0)>0
End
```

B.9 Scrolling text

If you have ever seen an Amiga demo then you will be familiar with scrolling text.

Here is a little routine which demonstrates scrolling text, Blitz Basic 2 style. The program copies a text message from the buffer (an area which cannot be seen by the user) onto the screen, one pixel at a time.

Try adding some clever effects to the text, such as diagonal scrolling. For the ultimate in text scrollers, why not incorporate the Mirrored Text routine so that the text is mirrored on the x-axis as it scrolls:

```
; *** Scrolling text
; *** Filename - Scroll_Text.bb2
; *** Author - Neil Wright

BLITZ
BitMap 0,384,270,3
BitMapOutput 0
Slice 0,44,320,256,$fff8,3,8,8,384,384
Use Palette 0
Show 0

; *** Change TEXT$ to anything you like
TEXT$="This is Neil Wright's fabulous text scrolling routine, "
TEXT$+"written completely in Blitz Basic 2!                  "
Colour 2

; *** Simple colour rainbow
For A=15 To 1 Step -1
  ColSplit 2,A,A,A,-3+A*2
Next A
B=1

; *** Main loop
Repeat
  Locate 43,1
  Print Mid$(TEXT$,B,1)
  Scroll 8,5,350,16,0,5
  VWait
  Let B+1
  If B=Len(TEXT$) Then B=1
Until Joyb(0)>0
```

B.10 Chipset?

If you are developing software for the Amiga series of computers then you may find it useful to know what chipset the host machine has. This small (but perfectly-formed) routine by Noel Baldacchino does just that:

```
; *** Chipset?
; *** Filename - Chipset.bb2
; *** Author - Noel Baldacchino

Function chipset{}
        MOVE.b  $dff07d,d0         ; *** ChipSet ID
        CMP.b   #$fc,d0            ; *** ECS?
        BNE.b   NoECS
        Function Return 0
NoECS:  CMP.b   #$f8,d0            ; *** AGA?
        BNE.b   NoAGA
        Function Return 1
NoAGA:  Function Return 2
End Function

R=chipset{}
Select R
  Case 0 : M$="You have an ECS machine! Upgrade now!"
  Case 1 : M$="Wow! You have an AGA machine!"
  Case 2 : M$="What! Why do you still have an OCS Amiga?"
End Select
NPrint M$
MouseWait
End
```

Well that's just about it on the programming front, folks; I hope you find the above programs useful. Now it is your turn - see if you can add some improvements to make the above programs better. As any hacker knows, the best way to learn about computing is to get lots of hands-on experience. Altering other people's programs is a good step on the way to producing your own programs completely from scratch - I would love to see what you come up with. Send your source code to the Blitz User Group, care of Neil Wright, at the address shown in Chapter 18. And do remember to include an SAE if you want your programs returned!.

Thank you for using my guide - happy programming, and I hope you enjoy a fruitful relationship with Blitz Basic 2!

Appendix C : Error Messages

C.1 You're bugging me

The two major stumbling blocks for computer programmers are bugs and errors. These may be typing mistakes made when you typed the program into the computer, or errors of logic in your code. Before you can get the program to work correctly you have to find all of the bugs and correct them.

When you are writing programs it often helps to remember that the computer can carry out three main activities: simple instructions, loops, and making decisions; these are the building blocks of all Blitz Basic programs. This guide has covered all of the instructions you need in Blitz Basic to tell the computer to carry out these activities.

There are usually several different ways to write a program and some of them may be shorter and faster than others. When you are writing a long program is it a good idea to divide it up into lots of sections with sub-routines or procedures to carry out each activity. Breaking up programs into sections like this makes it much easier to find any mistakes.

Usually it's a simple matter of correcting spellings and syntax. Blitz Basic does try to help by reporting errors as they occur, but wouldn't it be nice if we knew what these errors actually meant?

C.2 Blitz error messages

The bold text is the actual message that is reported by Blitz Basic when an error occurs. Underneath is a brief description of the problem, and/or the possible solution.

"**Already Included**"

The same source code has been previously included in the code.

Array already Dim'd"

A DIM statement is trying to dimension an array that has already been dimensioned. Arrays may not be re-dimensioned:

```
Dim A$(10)
;
Dim A$(10)
```

"**Array is not a List"**

A LIST function has been applied to an array that was not dimensioned as a List array.

"**Array not Dim'd"**

An undimensioned array has been unsuccessfully accessed. Check that the array names are correct.

"**Array not found"**

An array cannot be found in the main program.

"**Array not yet Dim'd**"

An undimensioned array has been unsuccessfully accessed. Check that the array names are correct.

"**Bad Data**"

The data following a DATA statement is of incorrect Type. Consult Chapter 1 for the correct Data Types.

"**Bad Type for For...Next**"

The FOR...NEXT index must be of numeric Type.

"**Can't Access Label**"

An undefined label has been unsuccessfully accessed. Check spelling of labels and GOTO/GOSUB branches.

"**Can't Assign Constant**"

Constant values cannot be assigned to variables.

"**Can't Assign Expression**"

The expression cannot be evaluated, or the evaluation has generated a value that is incompatible with the equate.

"**Can't Compare Types**"

The Type is incompatible with operations such as compares.

"**Can't Convert Types**"

The two Types are incompatible and one cannot be converted to the other.

"**Can't Create in Direct Mode**"

Variables cannot and must not be created in direct mode.

"**Can't create Macro inside Macro**"

Macro definitions cannot be created inside other macro definitions.

"**Can't Create Variable inside Dim**"

An undefined variable has been used as a dimension parameter with the DIM statement.

"**Can't Dim Globals in Procedures**"

Global variables cannot be defined within procedure definitions.

"**Can't Exchange different types**"

The EXCHANGE statement can only swap two variables of the same Type.

"**Can't Exchange NewTypes**"

The EXCHANGE statement cannot use NewTypes.

"**Can't Goto/Gosub a Procedure"**

Procedures cannot be branched to using GOTO/GOSUB.

"**Can't Load Resident"**

Blitz Basic cannot find the Resident file listed in the Options requester. Check the pathname.

"**Can't Nest Procedures"**

Procedures may not be defined within other procedure definitions.

"**Can't nest SetErr"**

Interrupt handlers cannot be nested.

"**Can't open Include"**

Blitz Basic cannot find the Include file; check the pathname.

"**Can't use comma in Let"**

The assigned variable is not a NewType or has only one entry.

"**Can't Use Constant"**

This is caused by clashing constant names.

"**Can't use Set/ClrInt in Local Mode"**

Error handling must occur in the primary code.

"**Case Without Select"**

A CASE statement has been found which does not have a corresponding SELECT statement.

"**CEND without CNIF/CSIF..."**

A CEND statement has been found which does not have a corresponding CNIF/CSIF statement.

"**Clash in Residents"**

Residents may not include the same Macro and Constant definitions.

"**CNIF/CSIF without CEND"**

A CNIF/CSIF statement has been found which does not have a corresponding CEND statement.

"**Constant already defined"**

Constants may only be defined once.

"**Constant not defined"**

An undefined constant has been used in an expression.

"**Constant Not Found"**

An undefined constant has been used in an expression.

"**Cont only Available in Direct Mode**"

CONT can only be called from Direct Mode.

"**Cont Option Disabled**"

The CONT option in the Options menu has been disabled.

"**Default without Select**"

A DEFAULT statement has been found which does not have a corresponding SELECT statement.

"**Direct Mode Buffer Overflow**"

Direct Mode has run out of memory. Try turning the "make smallest code"** option off.

"**Duplicate For...Next Error**"

The same index has been used within two nested FOR...NEXT loops:

```
For A=1 To 10
  For A=1 To 10
    Print " "
  Next A
Next A
```

"**Duplicate Label**"

The same label has been used more than once within a program.

"**Duplicate Offset (Entry) Error**"

The NewType has two entries with the same name.

"**Duplicate parameter variable**"

Parameters listed in Blitz Basic keywords must be unique.

"**Duplicate Procedure name**"

Procedures must be unique in name.

"**Duplicated Type**"

Types must be unique in name.

"**Element isn't a pointer**"

The variable used is not a VAR Type and cannot point to another variable.

"**End NewType without NewType**"

An END NEWTYPE statement has been found which does not have a corresponding NEWTYPE statement.

"**End Select without Select**"

An END SELECT statement has been found which does not have a corresponding SELECT statement.

"**End SetErr without SetErr**"

An END SETERR statement has been found which does not have a corresponding SETERR statement.

"**End SetInt without SetInt**"

An END SETINT statement has been found which does not have a corresponding SETINT statement.

"**Error Reading File**"

AmigaDOS has generated an error whilst reading a file from disk. Some of the data may be corrupt or missing.

"**Expression too Complex**"

This should never occur.

"**For...Next Block too Long**"

A FOR...NEXT loop has exceeded the Blitz limit of 32K in size. Try removing or repositioning any non-essential code.

"**For Without Next**"

A FOR statement has been found which does not have a corresponding NEXT statement. Make sure the FOR index matches the NEXT index:

```
; *** This is wrong
For A=1 To 10
Next B
```

"**Fractions Not allowed in Constants**"

Constants can only contain absolute values.

"**Garbage at End of Line**"

This usually happens when semi-colons are omitted from REMarks.

"**If Block too Large**"

An IF...ENDIF control structure has exceeded the Blitz limit of 32K in size. Remove any non-essential code from the structure.

"**If Without End If"**

An IF statement has been found which does not have a corresponding END IF statement. Both commands must be present:

```
A=1
If A>0
  Print "A is greater than zero"
  Mousewait
End If
```

"**Illegal Absolute"**

The Absolute location specified must be defined and in range.

"**Illegal Array type"**

This should never occur.

"**Illegal Assembler Addressing Mode"**

The addressing mode is not available for the opcode. Check the Blitz Basic Reference Manual for more information.

"**Illegal Assembler Instruction Size"**

The instruction size is not available. Check the Blitz Basic Reference Manual for more information.

"**Illegal Constant"**

Constant values may not be assigned to variables.

"**Illegal Constant Expression"**

Constants can only consist of integers.

"**Illegal direct mode command"**

Direct Mode cannot execute this command.

"**Illegal Displacement"**

The Displacement location specified must be defined and in range.

"**Illegal Else in While Block"**

Check Chapter 4 in this guide for the correct use of the ELSE statement within a WHILE...WEND block.

"**Illegal End Procedure"**

The procedure return syntax is incorrect. Check the relevant syntax in Chapter 4.

"**Illegal Function Type"**

A function may not return a NewType.

"**Illegal Immediate Value"**

An Immediate value must be a constant and in range. Check the Blitz Basic Reference Manual for more information.

"**Illegal Interrupt Number"**

Amiga Interrupts may range from zero to 13 only.

"**Illegal Label Name"**

Consult Chapter 1 for the correct use of variables.

"**Illegal Local Name"**

The variable name is not valid. Consult Chapter 1 for acceptable variable names.

"**Illegal number of Dimensions"**

List arrays are limited to single dimensions.

"**Illegal Operator for Type"**

The operator for a Type is incorrect.

"**Illegal Parameter Type"**

NewTypes cannot be passed to procedures.

"**Illegal Procedure Call"**

The procedure call syntax is incorrect. Check the relevant syntax in Chapter 4.

"**Illegal Procedure return"**

The procedure return syntax is incorrect. Check the relevant syntax in Chapter 4.

"**Illegal TokeJsr token number"**

A library routine referred to by the TOKEJSR statement cannot be accessed. This is usually caused by the absence of a specific library from DefLibs.

"**Illegal Token"**

This should never occur.

"**Illegal Trap Vector"**

The 68000 microprocessor has only 16 trap vectors.

"**Illegal Type"**

An illegal Type has been used in a function or statement.

"**Illegally nested Interrupts"**

Interrupt handlers cannot be nested.

"**Label has been used as a Constant**"

A label and a constant have been named the same; labels and constants cannot share the same name.

"**Label not Found**"

An undefined label has been unsuccessfully accessed. Check for any spelling mistakes in label names.

"**Label reference out of context**"

This should never occur.

"**Library not Available in Direct Mode**"

The library is not available in direct mode.

"**Library not Found : 'library number'**"

A library routine referred to by a token cannot be accessed. This is usually caused by the absence of a specific library from DefLibs.

"**Macro already Defined**"

Macros must be unique in name.

"**Macro Buffer Overflow**"

Increase the size of the macro buffer in the Options requester.

"**Macro not Found**"

An undefined macro has been unsuccessfully accessed.

"**Macro too Big**"

Macros are limited to the buffer size defined in the Options requester.

"**Macro without End Macro**"

A MACRO statement has been found which does not have a corresponding END MACRO statement.

"**Macros Nested too Deep**"

This should never occur.

"**Mismatched Types**"

This usually happens when string variables are given numeric values or vice versa. It can also arise when you try to EXCHANGE the values of string variables with numeric values.

"**Next without For**"

A NEXT statement has been found which does not have a corresponding FOR statement. Both commands must be present.

"**No Terminating Quote**"

All text strings should be enclosed in quote marks (e.g. "STRING").

"**Not Enough Parameters"**

A command has been given too few parameters. Check the relevant command syntax in this guide.

"**Not Supported"**

This should never occur.

"**Numeric Over Flow"**

The signed value is too large to fit in the provided variable space.

"**Offset not Found"**

The offset has not been defined in the NewType definition.

"**Only Available in Amiga mode"**

A command is only available in Amiga mode. Refer to the relevant command in this guide for correct mode details. If in doubt, try adding QAMIGA before the command:

```
; *** Go into a quick Amiga mode
QAMIGA
; *** Execute command
WaitEvent
; *** Return to Blitz mode
BLITZ
```

"**Only Available in Blitz mode"**

A command is only available in Blitz mode. Refer to the relevant command in this guide for correct mode details.

"**Optimizer Error! - $'"**

This should never occur.

"**Precedence Stack Overflow"**

This should never occur.

"**Previous Case Block too Large"**

The CASE section in a SELECT...END SELECT block has exceeded the Blitz limit of 32K in size. Try removing or repositioning any non-essential code.

"**Procedure not found"**

An undefined procedure has been called. Check for spelling mistakes and typing errors.

"**Repeat Block too large"**

REPEAT...UNTIL/FOREVER blocks are limited to 32K in length.

"**Repeat without Until**"

A REPEAT statement has been found which does not have a corresponding UNTIL statement.

"**Select without End Select**"

A SELECT statement has been found which does not have a corresponding END SELECT statement.

"**SetErr not allowed in Procedures**"

Error handling cannot be accessed in a procedure definition.

"**SetInt without End SetInt**"

A SETINT statement has been found which does not have a corresponding END SETINT statement.

"**Shared outside of Procedure**"

The SHARED statement must be contained within a procedure definition.

"**Syntax Error**"

There is a syntax error in the program. Either a command is not a Blitz reserved keyword or has an incorrect parameter - refer to the relevant syntax in this guide. This is an extremely common problem.

"**Token Not Found : 'token number'**"

An unknown token has been found. Check the spelling of all Blitz reserved keywords.

"**Too many comma's in Let**"

A NewType has fewer entries than the number of values in LET.

"**Too many parameters**"

A command has been given too many parameters. Check the relevant command syntax in this guide.

"**Type Mismatch**"

This usually happens when string variables are given numeric values or vice versa. It can also arise when you try to EXCHANGE the values of string variables with numeric values.

"**Type Not Found**"

An undefined Type has been unsuccessfully accessed.

"**Type too Big**"

The unsigned value is too large to fit in the variable space provided.

"**Unterminated Procedure**"

A procedure has been created but its definition has not been terminated. The relevant END FUNCTION or END STATEMENT commands must be present.

"**Until without Repeat**"

An UNTIL statement has been found which does not have a corresponding REPEAT statement. A REPEAT...UNTIL loop requires both statements to be present, otherwise an error is generated.

"**Variable already Shared"**

Variables can only be SHARED once.

Appendix D : Glossary

The programming world is littered with words which are exclusive to computing. Here is a list of some of the computer "jargon" used in this guide.

D.1 Glossary of terms

Address

An address is an integer number which identifies a memory location. All memory locations have different addresses.

AGA

The Amiga 1200 is built around the AGA (Advanced Graphics Architecture) custom chipset which can display up to 256 colours from a palette of 16.8 million, amongst other things. In Blitz Basic this translates to more colours, higher resolutions and wider sprites.

Amiga

The world's most powerful home computer. The Amiga was designed by a company called Amiga Incorporated and was produced by Commodore in 1985. At the time of writing (1995) the Amiga brand name is over ten years old!

Application

Software prepared for a specific function or set of functions, such as a paint package or map editor.

Array

An array is a list of variables of the same name that are distinguished by subscripts (values that identify each variable or element in the array).

ASCII

An acronym for American Standard Code for Information Interchange. A set of definitions for bit composition of characters and symbols. ASCII defines 128 symbols using seven binary digits and one parity bit.

BASIC

BASIC stands for Beginners All-purpose Symbolic Instruction Code. It uses an easily grasped mixture of English, numbers, strings, arithmetic signs and parameters which will enable you to start programming without having to learn a daunting low-level language such as Assembly Language. BASIC is a high-level language, as opposed to a low-level language such as machine code. Blitz Basic 2 is an optimised and extended dialect of BASIC.

Binary

Binary is a base two numeric system, in which all numbers are represented by the digits zero and one:

Table D.1 : Binary notation

```
Decimal: 1 2  3  4   5   6   7   8    9    10
Binary:  1 10 11 100 101 110 111 1000 1001 1010
```

BitMap

BitMaps are used primarily for rendering graphics. Most commands in Blitz Basic for generating graphics (excluding the Window and Sprite commands) depend upon a currently used BitMap.

Blitter

The Amiga's BLock Image TransfER device, or **Bit blatter", is used for copying large areas of memory from A to B, or to combine different areas into one single image. Widely used to generate Blitz Basic shape objects.

Blitting

Blitting is the name for the drawing of shape objects to BitMaps.

Blitz Basic

The world's most powerful BASIC for the Amiga range of computers. Blitz Basic has it all: speed, looks and a poor set of manuals. If Blitz Basic 2 was a car then it would be a Ferrari!

Blitzfont

Blitzfonts are used in the rendering of text to BitMaps. They must be eight-by-eight non-proportional fonts.

Buffer

A part of the computer's memory where data for input or output is held until it can be processed. Some makes of printer also have storage buffers.

Bug

A bug is a mistake, or error in a program. The process of removing these bugs is known as debugging.

B.U.G

B.U.G is a new (at the time of writing) club dedicated to all aspects of Blitz Basic 2. The aims of the club are simple: to teach people how to create commercial quality games in Blitz and make as much money from their programs as possible. Or something. Consult Chapter 18 for more information.

Byte

A unit of computer memory. Each individual byte can hold a single character or number from 0-255.

CLI

The Command Line Interface opens a window to communicate directly with AmigaDOS. It offers an alternative method of control to the Workbench icons.

Command

Commands are Blitz Basic tokens that can be used as either a function or a statement:

```
; *** Commands example
; *** Filename - Commands.bb2

ev.l=WaitEvent ; *** as a function
Waitevent      ; *** as a statement
MouseWait
End
```

Comments

Comments, or REMarks are lines in BASIC programs which are not executed. They are used to annotate programs, so as to make them easier to comprehend. In Blitz Basic comments must be preceded by a semi-colon:

```
; *** Comments example
; *** Filename - Comments.bb2

; *** Enter super-speedy Blitz mode
BLITZ
; *** Open Blitz mode display (3 bitplanes)
BitMap 0,320,256,3
Slice 0,44,3
Show 0
; *** Enable BitMap output
BitMapOutput 0
; *** This line prints "Hello"
NPrint "Hello"
; *** Wait for a mouse press
MouseWait
; *** End the program
End
```

Commercial games

Commercial, or full price, games are those available in the shops. Blitz Basic has been used to create a number of top commercial games, including Skidmarks (an isometric driving game with hundreds of frames of animation), Roadkill (an overhead driving game), and Worms (a weird puzzle game). Although some companies shy away from games created using BASIC, Acid Software, the publishers of Blitz Basic 2, don't. They love them!

Compiler

Compilers take programs written in a form that humans can understand and translate them into machine code, the simple, fast language of the processor chip inside the computer.

Co-ordinates

Co-ordinates are used to specify the position of a window, Slice, cursor, or sprite on the display. The X co-ordinate specifies the horizontal distance from the left-hand side of the screen, and the Y co-ordinate determines the vertical distance from the top of the screen. Both are measured in pixels.

Copper

The Copper (Co Processor) is used to generate subtly coloured backgrounds, or copper lists - this device, built into the Agnus chip, can alter colours while the screen is being generated. It can execute instructions at the same time as the main processor.

Cursor

The cursor is a marker which shows the area of the screen at which you are located. This is the position at which typed characters will appear. In Ted, the Blitz Basic editor, the cursor is a yellow block which marks the position in the program where you are working.

Cycling

Cycling in computer terms refers to the ability of colours in a colour palette to change place, or cycle. This primitive form of animation can be used to produce simple fades and psychedelic displays.

Data

Data describes information entered into or used by a computer.

Debugging

The process of removing bugs, or mistakes, from computer programs. When you are writing a long program it is a good idea to divide it up into lots of sections with sub-routines or procedures to carry out major activities. Breaking up programs into sections makes it much easier to find any mistakes.

Decimal

Decimal is the base ten number system in which all numbers are represented by the digits 0-9.

Directory

A directory, or drawer, is a structure on a disk. The space available on disks can be divided into a hierarchy of directories to allow the individual files to be split up into related categories.

Disk

Diskettes are used for storing and retrieving information, or data. Floppy disks are always spelt with a "k", nomatter what some people may tell you. Compact Discs are the exception to the rule.

Double buffering

Graphics are drawn on a hidden screen and copied to the displayed screen to create super-smooth displays.

Expression

An expression is a combination of constants, variables, and other expressions with operators. Expressions are evaluated by the interpreter to produce a string or numeric value.

Extra Half-Brite

Usually known as just "Half-Brite", this is a special display mode which doubles the number of colours on screen by dublicating the existing palette at half its brightness.

File

A file is a sequence of bytes which can be held in memory or stored to disk. These bytes can represent any type of data, such as pictures, samples, and music.

Floating Point Numbers

Floating Point Numbers are what the Mathematics world refers to as "Real numbers". They are numbers which can contain a decimal fraction as well as a whole number part (e.g. 16.17).

Fonts

In typography, a complete set of characters of the same size and style.

Fractal

A mathematical pattern created using recursion.

Function

Functions are Blitz Basic tokens that require parameters in parentheses, and return a value:

```
; *** Functions example
; *** Filename - Functions.bb2

N=Abs(-10)
MouseWait
End
```

Gadget

Gadgets are boxes which appear when the program requires you to enter or alter information. They are selected by clicking on the gadget once with the mouse pointer, although some gadgets require you to enter text (string gadgets).

HAM

Hold And Modify (HAM) is a special display mode which allows the full Amiga colour palette (4096 colours) to be displayed.

Handle

The single-pixel reference point of a graphical image. Commonly referred to as a "hot spot".

Hexadecimal

The hexadecimal system counts in units of 16 rather than ten, so a total of 16 different digits is needed to represent the different numbers. The digits from zero to nine are used as normal, but the digits from ten to 15 are signified by the letters A to F inclusive:

Table D.2 : Hexadecimal notation

```
Hex digit: 0 1 2 3 4 5 6 7 8 9 A  B  C  D  E  F
Decimal:   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Icon

Icons are visual representations of tools, projects, drawers, or disks.

IFF

IFF stands for Interchangeable File Format. Devised by Electronic Arts, it has been adopted as the standard way of storing pictures and sound on the Amiga.

Include files

These are files of predefined data which may be "included" in a source code file using a special directive. When the program is compiled these additional files are compiled as part of the main source code.

Integer

Integers are whole numbers (e.g. 10, 16 and 256), as opposed to floating point numbers, which do have a fractional part.

Interlace

A special display mode which doubles the vertical screen, or Slice, resolution.

Interrupt

Interrups are hardware signals which cause the Amiga's processor to stop what it is doing (usually the execution of the main program) and execute a pre-defined piece of code called an interrupt routine, or interrupt handler.

Keyboard shortcut

A method for performing a mouse action, such as the selection of a menu item, by pressing a key.

Keyword

The Blitz Basic 2 instruction set consists of a number of reserved keywords which perform a specific task. It includes the names of all Blitz Basic statements, functions, commands and operators. Examples include PRINT, EDIT$, WAITEVENT and <>.

Kilobyte (K)

The kilobyte is a unit of measurement of computer memory. One kilobyte is equal to 1,024 bytes. Although the name implies 1,000, the kilobyte is not, and never will be, 1,000 bytes - it is a corruption of the English language!

List arrays

List arrays differ from normal arrays in that Blitz Basic keeps an internal count of how many elements are stored in the List and an internal pointer to the current element within the List. List arrays are restricted in size to one dimension.

Macro

A single statement which can be used to represent a larger sequence of functions, statements or commands.

Mark Sibly

Father of Blitz Basic and all-round programming God. Mark hand-crafted Blitz using assembly language. Wow!

Menu

Menus are lists of items. You can see the titles of the menus available at a particular time by pressing the right-hand mouse button. Menu items are accessed by holding down the right mouse button and highlighting the correct item, before releasing the mouse button.

Menu item

An option which appears below a menu title.

NewType

In addition to the six primitive types available, programmers can also create their own custom types, or NewTypes. A NewType is a collection of fields, similar to a database or C structure, which enables you to group together relevant fields in one variable type. NEWTYPE must be followed by a list of entry names separated by colons and/or newlines:

```
; *** NewTypes example
; *** Filename - NewTypes.bb2

NEWTYPE .p
  X.w
  Y.w
  SPEED.w
End NEWTYPE

MouseWait
End
```

Palettes

Palettes, or palette objects, are temporary storage areas of colour information. This information can be taken either from an IFF file or created from scratch using Blitz Basic's palette functions.

Parallaxing

Parallaxing is a technique whereby parts of the display are scrolled at different speeds to create an illusion of depth. Because the television or monitor screen is 2-dimensional, all moving graphics appear flat. Parallax scrolling attempts to overcome the limits imposed by the screen, to produce a pseudo-3-dimensional display.

Parameter

A parameter is a piece of user-defined data which forms part of a Blitz Basic command. Parameters are used to control how commands operate:

```
; *** Parameters example
; *** Filename - Parameters.bb2

BLITZ
BitMap 0,320,256,3
Slice 0,44,3
Show 0
BitMapOutput 0
X=20
Y=10
; *** X and Y are parameters
Locate X,Y
NPrint "Moved, I'm sure"
MouseWait
End
```

Pixels

This is short for picture elements, allegedly. Pixels are the tiny graphical elements which make up the display. A standard, low-resolution PAL display has a resolution of 320 (horizontal) by 256 (vertical) pixels, or 81,920 pixels. By increasing the resolution, higher graphical definition can be achieved.

Procedure

A procedure is a specially defined module of code that can be called from your main program. Blitz Basic 2 supports two types of procedure: the function-type procedure and the statement-type procedure. A procedure which does not return a value is known as a statement and a procedure which does return a value is known as a function.

Proglet

A small snippet of code which demonstrates a basic principle or programming technique. Some programmers refer to these as routines, or pseudo-code, but I prefer proglets!

Program

A list of instructions which tell the computer to carry out a particular task or tasks. Blitz Basic is a program, and so is the code it creates.

Public Domain (PD)

Public Domain, or PD, describes the thousands of copyright-free disks that can be copied and sold by anybody. PD is non-profit making and as such very cheap to obtain. Blitz Basic 2 has been used to create some spectacular Public Domain software, including Zombie Apocalypse (an Operation Wolfstyle shoot-em-up), Defender (an update of one of the original arcade games), Insectoids (a vertically scrolling shoot-em-up similar to Galaxians), and Speed (a very slick card game).

Qualifier

Key which "qualifies", or changes the state of, a key-press. Examples include Shift, Ctrl, Alt and L Amiga/R Amiga.

Rainbow

A background colour graduation created using the COLSPLIT statement. Rainbows can provide a relatively good alternative to BitMap graphics in computer games.

RAM disk

A storage area which can be used to temprarily hold programs in memory for faster access than loading from disk.

Redraw

To redisplay the contents of a display. Technique used when the status of a gadget is altered.

Resolution

This describes the dimensions, in pixels, of a particular display mode. A low-res PAL screen has a resolution of 320 (horizontal) by 256 (vertical) pixels. A hi-res NTSC screen has a resolution of 640 (horizontal) by 200 (vertical) pixels.

Routine

An independent section of code which either works as part of the main program and can be reused again and again, or demonstrates a programming technique. See also "proglet".

Sample

Short sounds that can be played individually or as part of a song, or module. Samples are stored digitally in the Amiga's memory.

Scrolling

Scrolling is a technique whereby a display larger than the physical screen (a super-BitMap) is moved about one step at a time.

Slice

Slices are Blitz Basic objects which are the heart of Blitz mode's powerful graphics system. Through the use of Slices, many weird and wonderful graphical effects can be achieved, effects not normally possible in Amiga mode. This includes such things as dual playfield displays, smooth scrolling, double buffering and much more!

Speech

One of the fun utilities provided with the Amiga was the narrator device; this allowed pre-AGA Amigas to "talk". For reasons known only to themselves, Commodore chose to remove the "speech" facility from Workbench 3. A recent update has added speech to Blitz, so that owners of all Amigas (including those equipped with the AGA chipset) can access this fabulous facility through BASIC.

Sprites

Sprites are graphical elements which can be moved independent of the background. They are fast-moving but are restricted in size, colour and number.

Stack

The stack is an area of memory which is used by Blitz Basic for temporary information storage.

Statement

Statements are Blitz Basic tokens that perform an action but do not return a value. Their arguments do not require parentheses:

```
; *** Statements example
; *** Filename - Statements.bb2

NPrint "Blitz Basic 2"
MouseWait
End
```

String

A string variable is one which contains text, rather than numbers. Strings are surrounded by quotation marks and all string names must end with the dollar ($) character.

Sub-routine

A sub-routine is a section of code that is separate to the main program. A sub-routine is a sort of mini-program within a program. It carries out a particular task, such as updating the display, or controlling object movement. All sub-routines are preceded by a program label and may be called using the GOTO and GOSUB statements.

Title bar

The optional top border of a screen or window, which displays the screen and window titles respectively.

Toggle

An option, such as a gadget, which can be toggled between two states (usually on and off).

Tracker

A Tracker is a sequencing program which allows you to enter musical motes and arrange them to create a song, or module. The best Tracker program on the Amiga is the commercial OctaMed program.

Types

Blitz BASIC currently supports six different types of variable, five numeric types for numeric data and one string type ($) for strings:

Table D.3 : Blitz Basic Types

Type	Suffix	Range	Accuracy	Bytes	Example
Byte	.b	+/- 128	Integer	1	Neil.b=125
Word	.w	+/- 32768	Integer	2	Dan.w=30000
Long	.l	+/- 2147483648	Integer	4	Jon.l=$dff000
Quick	.q	+/- 32768.0000	1/65536	2	Richard.q=500/7
Float	.f	+/- 9e18	1/10e18	4	Craig.f=4e7

Variable

Variables are numeric pointers used to store pieces of information. A variable containing numbers is called a numeric variable and one which contains letters and symbols is known as a string variable. Values can be assigned to variables as follows:

```
; *** Variables example
; *** Filename - Variables.bb2

Let A=5
; *** Assign the value 5 to variable A
NPrint A
; *** Print the contents of variable A
MouseWait
End
```

Venus

The second-closest planet to the sun, Venus has an oven-hot surface temperature of over 470 degrees Celsius and a diameter of nearly 7,500 miles. Venus is a happening planet!

Window

A rectangular area of the screen that can accept or display information. A window may have an optional title bar and/or gadgets in its border.

Workbench

Workbench is the Amiga's icon-based "point and click" Graphical User Interface. There are three main Workbenches in popular usage: Workbench 1.3 (Amiga 500), Workbench 2.X (Amiga 500+/600), and Workbench 3.X (Amiga 1200/4000).

WYSIWYG

What You See Is What You Get, or WYSIWYG is a term used in word processing and desk-top publishing which refers to the screen display, relative to the final output. WYSIWYG does not feature in this guide in any shape or form, so what you see is definitely not what you get!