BLITZ BASIC 2.1

Manual edited and improved by
Ivan 'Zeb' Elwood & Chris Forrester

Version 0.17 on 21st January, 2020

Original OCR scan by Mikael Mick Norrgård

Please report any errors to zebby@gmx.com

Directory Tree for Hard Disk Users

```
Blitz                                  program files
   |
   +--Blitzlibs                        resident files
   +--amigalibs                        amiga libraries
   +--otherlibs                        third party libraries
   |
   +-Developers/                       documentation
   | +--acidlibsrc                     acid library source code
   | +--amigaincludes                  system includes
   | +--toolsource                     developer toolsource code
   | +--userlibdocs                    docs for third party libs
   | +--userlibprogs                   test programs for third party libs
   | +--Userlibsource                  source code for third party libs
   |
   +-Examples/                         example code
   | +--amigamode
   | +--andrewsdemos
   | +--blitzmode
   | +--marksdemos
   | +--simonsdemos
   | +--tedsdemos
   | +--tools
   |
   +-Userlibs/
```

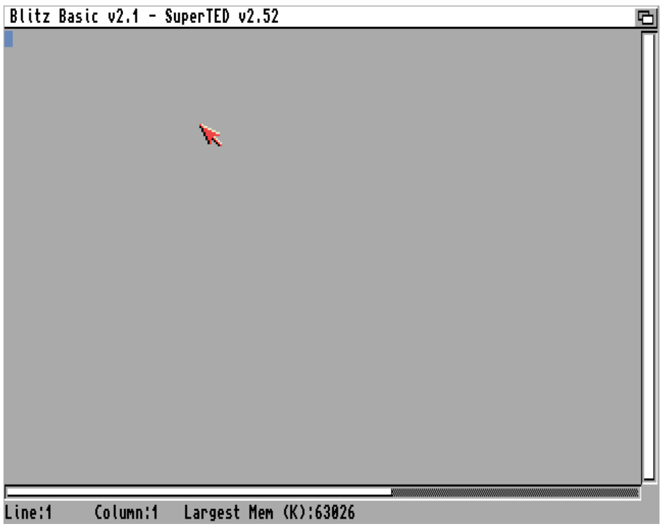
Using Ted the Blitz2 Editor

To enter and compile your programs you need an editor. Blitz2 comes with a text editor that acts both as an interface to the Blitz2 compiler as well as a stand-alone ASCII editor (ASCII is the computer standard for normal text).

The horizontal and vertical bars are called 'scroll bars'. When the file you are editing is longer or wider than the screen you can position your view of the file by dragging these bars inside their boxes with the left mouse button.

At the bottom of the screen is information about the cursor position relative to the start of the file you are editing as well as a memory monitor that lets you know the largest block of memory available in your Amiga system.

Using the left mouse button you can drag the Blitz2 screen up and down just like any other Amiga screen as well as place it to the back with the front to back gadgets at the top right of the screen.



```
Line:1     Column:1   Largest Mem (K):63026
```

Entering Text

The editor can be treated just like a standard typewriter; just go ahead and type, using the return key to start a new line.

The small box that moves across the screen as you type is called the cursor. Where the cursor is positioned on the screen is where the characters will appear as you type. By using the arrow keys you can move the cursor around your document, herein to be known as the file.

If you place the cursor in the middle of text you have already typed you can insert characters just by typing. The editor will move all the characters under and to the right of the cursor along one and insert the key you pressed into the space created.

The DEL key will remove the character directly under the cursor and move the remaining text on the line left one character to fill the gap.

The key to the left of the DEL key will also remove a character but unlike the DEL key it removes the character to the left of the cursor moving the cursor and the rest of the line to the left.

The TAB key works similar to a typewriter, moving the cursor and any text to the right of the cursor to the next tab stop.

The RETURN key, as mentioned, allows you to start a new line. If you are in the middle of a line of text and want to move all text to the right of the cursor down to a new line use shift RETURN, this is known as inserting a carriage return.

To join two lines of text use the Amiga J keyboard combination.

Using the shift keys in combination with the arrow keys you can move the cursor to the very start or end of a line and up and down a whole page of the document.

By pointing with the mouse to a position on the screen you can move the cursor there by clicking the left mouse button.

See keyboard shortcuts at the end of this chapter for other keys used with the Blitz2.

Highlighting Blocks of Text

When editing text, especially programs, you often need to operate on a block of text. Position the mouse at the start or end of the block, hold down the left mouse button and drag the mouse to highlight the area you wish to copy, delete, save or indent. While holding down the button you can scroll the display by moving the pointer to the very top or bottom of the display.

You can also select a block with the keyboard, position the cursor at the start of the block of text, hit the F1 key then position the cursor at the end of the text and hit F2. A special feature for structured programmers is the Amiga-A key combination, this automatically highlights the current line and lines any above or below that are indented the same number of spaces.

The Editor Menus

Using the right mouse button you can access the menu system of the Blitz2 editor. Following is a list of features accessible from these menus in order from left to right.

The PROJECT Menu

NEW

Kills the file you're editing from the Amiga's memory. If the file has been changed since it was last saved to disk a requester will ask you if you really wish to NEW the file.

LOAD

Reads a file from disk. A file requester appears when you select LOAD which enables you to easily select the file you wish to edit. See later in this chapter for a full description of using the file requester.

SAVE

Writes your file to disk. A file requester appears when you select SAVE which enables you to easily select the file name you wish to save your file as. See later in this chapter for a full description of using the file requester.

DEFAULTS

Changes the look of the Blitz2 editor. You can edit the palette, select the size of font and tell the system if you wish icons to be created when your files are saved. The scroll margins dictate how far from the edge of the screen your cursor needs to be before Blitz scrolls the text for you.

ABOUT

Displays the version number and credits concerning Blitz2.

PRINT

Sends your file to an output device, usually PRT:, the printer device.

CLI

Launches a command line interface from the editor. Use the ENDCLI command to close this CLI window and return to the Blitz2 editor.

CLOSE WB

Closes Workbench if it is currently open. This option should only be used if you are running very short on memory as closing Workbench can free about 40K of valuable chip mem.

QUIT

Close the Blitz2 editor and returns you to Workbench or CLI.

The EDIT Menu

COPY

Copies a block of text that is highlighted with the mouse or F1/F2 key combination to the current cursor position. The F4 key is another keyboard shortcut for COPY.

KILL

Deletes a highlighted block of text (same as shift F3 key).

BLOCK TO DISK

Saves a highlighted block of text to disk in ASCII format.

INSERT FROM DISK

Loads a file from disk and inserts it into the file you are editing at the cursor position.

FORGET

De-selects a block of text that is selected (highlighted).

INSERT LINE

Breaks the line into two lines at the current cursor position.

DELETE LINE

Deletes the line of text the cursor is currently located on.

DELETE RIGHT

Deletes all text on the line to the right of the cursor.

JOIN

Places the text on the line below the cursor at the end of the current line.

BLOCK TAB

Shifts all highlighted text to the right by one tab margin.

BLOCK UNTAB

Shifts all highlighted text to the left by one tab margin.

The SOURCE Menu

TOP

Moves the cursor to the top of the file.

BOTTOM

Moves the cursor to the last line of the file.

GOTO LINE

Moves the cursor to the line number of your choice.

The SEARCH Menu

FIND

Will search the file for a string of characters.

NEXT

Positions the cursor at the next occurrence of the find string entered using the FIND menu option (as below).

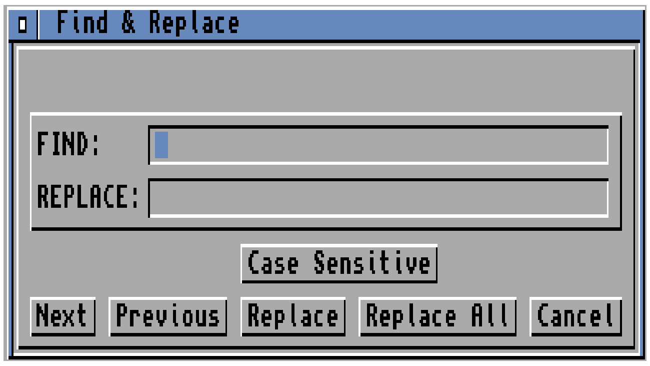
PREVIOUS

Will position the cursor at the last occurrence of the find string entered using the FIND menu option (as below).

REPLACE

Will carry out the same function as discussed in the FIND requester below.

After selecting FIND in the SEARCH menu the following requester will appear:



Type the string that you wish to search for into the top string gadget and click on NEXT. This will position the cursor at the next occurrence of the string, if there is no such string the screen will flash.

Use the PREVIOUS icon to search backwards from the current cursor position.

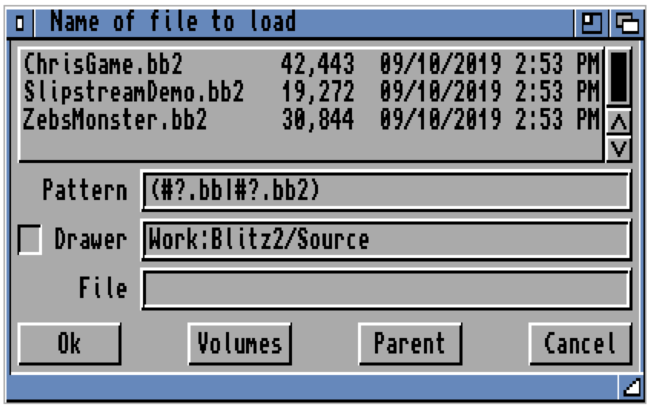
The CASE SENSITIVE option will only find strings that have same letters capitalised, default is that the search will ignore whether letters are caps or not.

To replace the find string with an alternate string click on the box next to REPLACE and type the alternate string. REPLACE will search for the next occurrence of the find string, delete it, and insert the replace string in its place.

REPLACE ALL will carry on through the file doing replaces on all occurrences of the find string text.

The Blitz File Requester

When you select load or save, Blitz2 places a file requester on the screen. With the file requester you can quickly and easily find the file on a disk.



```
ChrisGame.bb2        42,443  09/10/2019 2:53 PM
SlipstreamDemo.bb2   19,272  09/10/2019 2:53 PM
ZebsMonster.bb2      30,844  09/10/2019 2:53 PM
```

```
Pattern (#?.bb|#?.bb2)
Drawer  Work:Blitz2/Source
File
```

Clicking on the top left of the window or on the CANCEL gadget at the bottom right will cancel the file requester returning you to the editor.

The slider at the right enables you to scroll up and down through the files in the currently selected directory (drawer).

Double clicking on a file name (pointing to the name and pressing the left mouse button twice) will select that file name.

Clicking on PARENT will return you to the parent directory.

Clicking on DRIVES adds a list of all drives, volumes and assigned devices to the top of the file list so you can move into their directories.

You can also enter path and file names with the keyboard by clicking on the boxes next to PATH: and FILE: and entering the suitable text. Then click on the OK gadget.

CD is a special command used when programming in Blitz2 to change the editors current directory to that specified in the path name. When you select CLI or launch a task from the editor its root directory will be that selected by the CD gadget.

Last feature of Blitz2 FileRequester is the ability to size its window, dragging bottom-right of the window with the left mouse button you can see more files at one time.

The COMPILER Menu

The following is a discussion of the extra options and commands available with Ted when used in Blitz2 programming mode. The Compiler menu includes all the commands needed to control the Blitz2 compiler.

COMPILE/RUN

Compiles your Blitz2 program to memory and if there are no errors, run the program.

RUN

Runs the program if it has already been successfully compiled to memory.

CREATE FILE

Compile your Blitz2 program to disk as an executable program.

OPTIONS

See next page for details about Blitz2 compiler options.

CREATE RESIDENT

Will create a 'resident file' from the current file. A resident is a file including all constants and macro definitions as well as newtype definitions. By removing large chunks of these definitions from your code and creating a resident (pre-compiled) file a dramatic increase in compile speed can be obtained.

VIEW TYPE

Allows you to view all currently resident types. Click on the type name and its definition will be shown. Subtypes can be viewed from this expansion also.

CLI ARGUMENT

Enables you to pass parameters to your program when executing it from the Blitz2 editor environment just as if you had run the program from the CLI.

CALCULATOR

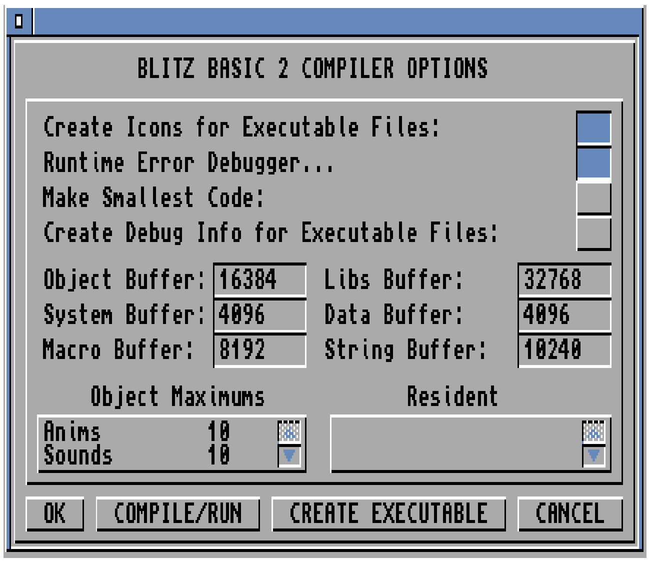
Allows you do to calculations in base 2, 10 and 16. Precede hex values with $ and binary with %. It also supports multi levels of parenthesis.

RELOAD ALL LIBS

Will read all files from BLITZLIBS: back into Blitz2 compiler environment. It's useful when writing your own Blitz2 libs and wish to test them without having to re-run Blitz2.

Compiler Options

The following is a discussion of the Options requester found in the Compiler menu.


BLITZ BASIC 2 COMPILER OPTIONS

Create Icons for Executable Files: ✓
Runtime Error Debugger... ✓
Make Smallest Code:
Create Debug Info for Executable Files:

Object Buffer:	16384	Libs Buffer:	32768
System Buffer:	4096	Data Buffer:	4096
Macro Buffer:	8192	String Buffer:	10240

Object Maximums

Anims	10
Sounds	10

Resident

Create Icons for Executable Files

If on, the compiler creates an icon to accompany the file created with the CREATE FILE option. This means the program will be accessible from the WB. Note: for the program to execute correctly when run from WB the WBStartUp command should be included at the top of the source code.

Enable Runtime Errors

When on will trap runtime errors and invoke the Blitz2 debugger. See Chapter 5 for a thorough discussion of runtime errors in Blitz2.

Make Smallest Code

Selects two pass compile mode, which always calculates the minimum amount of memory required for the object code. Make Smallest is automatically selected when creating executable files. Unselected, programs will compile quicker.

Debug Info

Creates a symbols table during CREATE FILE so executable can be debugged more easily with debuggers such as Metadigm's excellent MetaScope.

Buffer Sizes

Allows different buffers to be altered when using Blitz2 as a one-pass compiler. These buffers are automatically optimised when using Make Smallest (two-pass compile). The one exception is the string buffer setting. If using large strings (such as reading entire files into one string) the string workspace buffer should be increased in size to handle the largest string used.

Object Maximums

Allows setting of maximum number of Blitz2 objects such as screens, shapes etc. See Chapter 6 for a thorough explanation of Blitz2 objects and their maximum settings.

Resident

Adds pre-compiled resident files to the Blitz2 environment. Click in the box and type in the resident file name.

Keyboard Shortcuts

Having to reach for the mouse to execute some of the editor commands can be a nuisance. The following is a list of keyboard shortcuts that execute the same options that are available in the menus.

The right Amiga key is just to the right of the space bar and should be used like the shift key in combination with the stated keys to execute the following commands:

Amiga A Selects all text that is indented the same amount as the current line (strictly for structured programming housekeeping)
Amiga B BOTTOM will position cursor on last line of file
Amiga D DELETE LINE removes the line of text on the cursor position
Amiga F FIND/REPLACE executes the FIND command in the SEARCH menu
Amiga G GOTO LINE moves cursor to specific line of file
Amiga I INSERT LINE moves all text at and below the cursor down one line
Amiga J JOIN LINE adjoins next line with current line
Amiga L LOAD reads a file from disk
Amiga N NEXT searches for the next occurrence of the 'find string'
Amiga P PREVIOUS searched for previous occurrence of the 'find string'
Amiga Q QUIT will exit the Blitz2 editor
Amiga R REPLACE will replace text at cursor (if same as find string) with the alternate string specified with the Find command
Amiga S SAVE writes a file to disk
Amiga T TOP moves the cursor to the top of the file
Amiga W FORGET will unhighlight a selected block of text
Amiga Y DELETE TO RIGHT of cursor
Amiga Z CLI
Amiga ? DEFAULTS allows the user to change the look and feel of the Blitz2 editor
Amiga] BLOCK TAB moves whole block right one tab
Amiga [BLOCK UNTAB moves whole block left one tab

Type in the following two lines:

```
Print “This is my first program written in Blitz2!”
MouseWait
```

Then using the right button select COMPILE & RUN from the top right menu.

If you have typed the program in correctly a Blitz2 CLI Window will appear with the message. Click the mouse button to return to the editor. That's all there is to it!

The Print Command

Position the cursor on the Print statement and press the HELP key, the syntax for the Print command appears at the top of the screen. It should read:

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

The square brackets mean that the Print command will accept any number of expressions separated by commas. An expression can be any number, string (text in "quotes"), variable or BASIC calculation. The following is an example of all these.

Don't forget to include the MouseWait command when you test this, otherwise Blitz2 will print the message and return you to the editor before you even have time to read it.

```
Print 3,”CARS”,a,a*7+3
```

The following should be printed out in the CLI window: 3CARS03

If we add some spacing between each expression like so:

```
Print 3,” CARS ”,a,” “,a*7+3
```

The result will be the line: 3 CARS 0 3

Formatted Printing

We can change the way Blitz2 prints numbers using the Format command, this is useful if you want to print a list of numbers, in a column.

NPrint is used to move the cursor to a newline after printing the expressions.

```
Format “###.00”
NPrint 23.5
NPrint 10
NPrint .5
NPrint 0

MouseWait
```

A Simple Variable

The main power of a programming language lies in its ability to manipulate numbers and text. Variables are used to store these pieces of information.

The following line will store the value 5 in the variable a:

```
a=5
```

The variable a now holds the value 5. We can tell the computer to add 1 to the value of a making it 6 using the following expression:

```
a=a+1
```

An expression can contain more than one operation, brackets can be used to make one operation be evaluated before the others:

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

Blitz2 Operators

An evaluation is a collection of variables, constants, functions and operators. Examples of operators are the plus and minus signs.

An operator will generate an outcome using either the variable on its right: `a=NOT 5`

Or from the variables on its left and right:

```
a=5+2
```

An evaluation can include multiple operators:

```
a=5*6+3
```

As in mathematics the order the operators are evaluated will affect the outcome, if the multiply is done first in the above example the result is 33, if the addition was done first, 5*(6+3), the result will be 40.

When Blitz performs an evaluation some operators have precedence over others and will be evaluated first, the following two evaluations will have the same result because Blitz2 will always evaluate multiplication before addition, a=5*6+3 is the same as a=3+5*6

To override the order which Blitz2 evaluates the above, parenthesis can be added, operations enclosed in parenthesis will be evaluated first:

```
a=5*(6+3)
```

The following table lists the Blitz2 operators grouped in order of priority (LHS=left hand side, RHS=right hand side). Operators in the same box have the same priority.

NOT -	RHS logically NOT'ed RHS arithmetically negated
BITSET BITCLR BITCHG BITTST	LHS with RHS bit set LHS with RHS bit cleared LHS with RHS bit changed True if LHS bit of RHS is set
^	LHS to the power of RHS
LSL ASL LSR ASR	LHS logically shifted left RHS times LHS arithmetically shifted left RHS times LHS logically shifted right RHS times LHS arithmetically shifted right RHS times
& |	LHS logically AND'ed with RHS LHS logically OR'ed with RHS
* /	LHS multiplied by RHS LHS divided by RHS
+ -	LHS added to RHS RHS subtracted from LHS
= <> < > <= >=	True if LHS is equal to RHS True if LHS is not equal to RHS True if LHS is less than RHS True if LHS is greater than RHS True if LHS is less than or equal to RHS True if LHS is greater than or equal to RHS
AND OR	LHS logically AND'ed with RHS LHS logically OR'ed with RHS

Boolean Operators

The boolean system can only operate with two values, true and false. In Blitz2 false is represented by the value 0, true with the value -1.

The operators =, <>, <=, =>, > and < all generate a boolean result (true or false).

NPrint 2=2 will print value -1 as the result of the operation 2=2 is true. The operators OR, AND and NOT can be used as boolean operators, NPrint 2=2 AND 5=6 will print 0 as the result is false. OR operator returns true if either left or right hand side is true. NOT operator returns false if the following operand is true and true if the operand is false.

Binary Operators

Many of Blitz2 operators perform binary type arithmetic. These operations are very fast as they directly correspond to instructions built into the computer's processor.

The binary system means that all numbers are represented by a series of 1s and 0s. A byte is made up of X such bits, a word 16 and a long word 32.

Multiple Commands

The following program starts a with a value of 0, it then proceeds to add 12 to the value of a and print the result 4 times.

```
a=0
a=a+12:NPrint a
a=a+12:NPrint a
a=a+12:NPrint a
a=a+12:NPrint a

MouseWait
```

Note how we can put two commands on the same line by separating each command with a colon character. Also, the first line a=0 is not needed as variables in Blitz2 always start out with a value of 0 anyway.

A Simple Loop

The following program prints out the 12 times table. Instead of typing in 12 lines to do this we use a For...Next loop. A loop is where the program is told to repeat a section of the program many times.

For i=1 To 12...Next will execute the commands between the For and Next 12 times, the variable i is used to keep count.

The asterisk * means multiply so a=i*12 means the variable a now equals 12 x the variable i. Because i is counting up from 1 to 12 the variable a is assigned the values 12, 24, 36, 48... as the program loops.

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

MouseWait
```

Note how the two lines inside the loop are indented across the page. This practice makes it easy to see which bits of the program are inside loops and which are not.

The Tab key can be used to move the cursor across the page so many spaces when typing in lines that are indented.

Now try changing the first line to For i=1 To 100, as you can see the computer has no problem what so ever doing its 12 times table!

We could also change the number 12 in the first 3 lines to any other number to generate an alternative times table.

Nested Loops

The following program is an example of nesting loops, a term that refers to having loops inside of loops. By indenting the code that is inside inner loop even further we can keep a check to make sure each For statement lines up with each Next statement.

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

MouseWait
```

The nesting of the For x=1 To 12 inside the For y=1 To 12 means the line inside the For x will be executed 12 x 12 times, each time with a new combination of x and y.

While...Wend and Repeat...Until

There are two other simple ways to program loops in Blitz2 besides using For...Next. While...Wend and Repeat...Until loops are used as follows:

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

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

As with a lot of BASIC commands they are pretty much self explanatory, the inside of a While...Wend will be repeated while the condition remains true. A Repeat...Until will loop until the condition is true.

A condition can be any evaluation such as While a+10<50, While f=0, While b<>x*2 and so-on.

The difference between the two loops above is that if a was greater than 20 to start with, the Repeat...Until would still execute the code inside the loop once, where as the While...Wend would not.

Endless Loops

When a program gets into the situation of repeating a loop forever it is called an endless loop. In this situation the programmer must be able to override the program and tell it to stop.

To interrupt a program the Ctrl/Alt C keyboard sequence must be used. Holding down the Ctrl key and the Left Alt key press C, this will stop the program and the debugger screen will appear. To exit from the debugger and return to the editor use the Esc key (top left of the keyboard). The debugger is covered in detail in Chapter 5.

Using String Variables

Variables that contain text are called string variables. String variables require $ sign after their names. Following shows a simple example of a string variable:

```
a$=”Chris”
NPrint a$
MouseWait
```

Similar to numeric variables the = sign is used to assign the string variable a value. The + sign can be used to add strings together (concatenate):

```
a$=”Chris “:b$=”Forrester”:c$=a$+b$
```

The variable c$ will now contain the string “Chris Forrester”. Other functions that manipulate strings are detailed in the reference section of this manual.

Program Flow

Often a program will have to decide to do either one thing or another, this is called program flow. The If Then commands are used to tell the program to do something only if some condition is true. The following will only print "Hello" if the variable a has the value 5:

```
If a=5 Then Print “Hello”
```

The above line could be changed to do a section of commands if a was equal to 5 using the IF...Endif structure:

```
If a=5
  Print “Hello”
  a=a-1
Endif
```

The Else command is used to execute an alternative section if the condition is not true:

```
If a=5
  Print “Hello”
Else
  Print “Good-bye”
Endif
```

Note how we indent code inside conditional blocks just like we did with loops. This makes the code more readable, it is easier to see which lines of code will be executed when the condition is true etc.

The condition after the If command can be any complex expression, the following are some examples of possible test conditions:

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

An appendix at the end of this manual contains a complete description of using multiple operators and their precedence.

Jumpin’ Around

Often the program will need to jump to a different section of the code. The Goto and Gosub routines are used for this. The location that the program is jumping to needs a label so that Goto and Gosub can reference the location they are jumping to. The following uses the label start:

```
Goto start
NPrint “HI THERE”
start

MouseWait
```

The Goto statement makes the program jump to label start and “Hi There” is never printed. The Gosub command is used to jump to a subroutine. This is a piece of code terminated with a Return statement. This means that after executing the subroutine, program flow returns to where the Gosub command was executed and carries on.

```
.start:
  Gosub message
  Gosub message
  Gosub message
  MouseWait
  End

.message
  NPrint “Hello”
  Return
```

Note how labels are preceded with a period. This makes them appear in a list on the right of the editor screen. We can make the cursor jump to a label by clicking it in this list. This is extremely useful for when editing large programs.

Getting Input from the User

A program will often require input from the user when it is running either via the keyboard or mouse. For instance, the MouseWait command will stop the program until the user clicks the left mouse button.

Keyboard input can be obtained using the Edit and Edit$ functions which is the same as the Input command in other languages.

The following asks the user for their name and places it into a string variable:

```
Print “What is your name?”
a$=Edit$(80)
NPrint “Hello “,a$
MouseWait
```

Number 80 in Edit$(80) refers to maximum number of characters the user can type. To input numbers from the user the Edit function is used, a=Edit(80) will let the user type in any number up to 80 digits long and will place it in the variable a.

Arrays

Often a program will need to manipulate groups of numbers or strings. An array is able to hold such groups. If we needed to keep track of ten numbers that were all related, instead of using ten different variables we can define an array to hold them.

The Dim statement is used to define an array:

```
Dim a(10)
```

The variable a can now hold 10 (actually 11) numbers and to access them we place an index number inside brackets after the variable name:

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

The power of an array is that the index number can be a variable, if i=2 then a(i) refers to the same variable, is a(2).

The following inputs 5 strings from the user using a For...Next loop. Because the strings are placed in an array they can be printed back out:

```
Dim a$(20)

NPrint “Type in 5 names”
For i=1 To 5
  a$(i)=Edit$(80)
Next

NPrint “The names you typed were”
For i=1 To 5
  NPrint a$(i)
Next

MouseWait
```

3. TYPES, ARRAYS AND LISTS

Numeric Types

Blitz2 supports 6 different types of variables and there are 5 numeric types for storing numeric values with differing ranges and accuracies as well as a string type used to store strings of characters (text).

The following table describes each Blitz2 numeric variable type with details on its range and accuracy and how many bytes of memory each requires.

Type	Suffix	Range	Accuracy	Bytes
Byte	.b	+-128	integer	1
Word	.w	+-32768	integer	2
Long	.l	+-2147483648	integer	4
Quick	.q	+-32768.0000	1/65536	2
Float	.f	+-9*10/\18	1/10/\18	4

The Quick type is a fixed point type, less accurate than floating point but faster.

The Float type is the Floating Point type supported by the Amiga last FP libraries.

A variable is assigned a certain type by adding the relevant suffix to its name after the first reference to a variable, its type is assigned and any future references do not require the suffix unless it is a string variable.

Following are some examples of typical numeric variables with their relevant suffix.

```
mychar b=127
my_score.w=32000
chip.l=$dff000     ;$ denotes a hex value
speeD3.q=500/7     ;a quick has 3 decimal place accuracy
light_speed.f=3e8  ;e is exponent i.e. 3X10A8
```

Default Types

If no suffix is used in the first reference of a variable, Blitz2 will assign that variable with the default type. This is initially the Quick type.

There are two forms of the DefType command, one which changes the default type the other which defines the type of a list of variables supplied but which does not affect the default type.

The following code illustrates both uses of DEFTYPE:

```
a=20            ;a will be a quick
DEFTYPE .f      ;vars without suffix will now default to float
b=20            ;b will be a float
DEFTYPE .w c,d  ;c & d are words, default still float
```

Note: the second instance of DEFTYPE should be read define type rather than its first use which stands for change default type. The default type can also be set to a newtype (see the following section).

Other Blitz2 structures that work with certain type such as data statements, functions, peeks and pokes will also use default type if no type suffix is included.

The Data Statement

Used to hold a list of values that can be read into variables.

The Restore command is used to point the data pointer at a certain Data statement.

A type suffix is added to data statement to define what type the values listed are.

The following is an example of using Data in Blitz2:

```
main:
  Read a,b,c
  Restore myfloats
  Read d,f
  Restore mystrings
  Read e$,f$,g$
myquicks:
  Data 20,30,40
myfloats:
  Data.f 20.345,10.7,90.111
mystrings:
  Data$ “Hello”,”There”,”Simon”
```

Note: If the data pointer is pointing to a different type than the variable listed in the Read statement a Mismatched Types runtime error occurs.

Numeric Overflow & Unsigned Integers

When a variable is assigned a value outside of its range (too large), an overflow error will occur. The following code will cause an overflow error when it is executed:

```
a.w=32767          ;a is a word containing the number 32767
a=a+1              ;overflow occurs as result is out of range
```

Overflow checking is optional and can be enabled/disabled in the runtime errors options of the Compiler Configuration. The default setting is off meaning the above code will not generate a runtime error. In some instances, the integer types will be required to represent unsigned (positive only) numbers. For example, a byte variable will be required to hold values between 0 and 255 rather than -127 to 128. Overflow checking has to be disabled in the Error Checking requester of the Compiler Options window to use unsigned ranges such as this.

String Types

A string is a variable that is used to store a string of characters, usually text. The suffix for a string variable is either a .s or the traditional $ character.

Unlike numeric variables the suffix must always be included with the name. Also, string variable names MAY be re-used as numeric variable names.

The following is quite legal:

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

System Constants

Blitz2 reserves a few variables that hold special values known as system constants. The following variables are reserved and contain the listed values:

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

Primitive Types Summary

Blitz2 currently supports 6 primitive types. Byte, word and long are signed 8, 16 and 32 bit variable types. The quick type is a fixed point type, less accurate than FP but faster. The float type is the FP type supported by the Amiga fast FP libraries.

The string type is a standard BASIC implementation of string variable handling.

Using DefType directive variables can be defined as certain types without adding the relevant suffix. Once a variable is defined as a certain type, the suffix is not necessary, except in the case of string variables when the suffix must always be included.

A variable can only be of one type throughout the program and cannot be defined as any other except again in the case of strings where the variable name can ALSO be used for a numeric type.

NewTypes

In addition to the 6 primitive types available in Blitz2, programmers also have available the facility to create their own custom types.

A NewType is a collection of fields, similar to a record in a database or a C structure. This enables the programmer to group together relevant fields in one variable type.

The following code shows how fields holding a person's name, age and height can be assigned to one variable:

```
NEWTYPE .Person
  name$
  age.b
  height.q
End NEWTYPE

a.Person\name="Zeb",20,2.1
NPrint a\height
```

Once a NewType is defined, variables are assigned the new type by using a suffix of .NewTypename for example a.Person

Individual fields within a NewType variable are accessed and assigned with the backslash character '\' for example: a\height=a\height+1.

When defining a NewType structure, field names without a suffix will be assigned the type of the previous field. More than one field can be listed per line of a NewType definition but they must however be separated by colons. The following is another example of a NewType definition:

```
NewType .name
  x.w:y:z      ;y & z are also words (see above)
  value.w
  speed.q
  name$
End NewType
```

References to string fields when using NewTypes do not require $ or .s suffix as normal string variables do, including suffix will cause Garbage at End compile time error.

From the first example:

```
a\name="Jimi Hendrix"   ;this is cool
a\name$="Bob Dylan"     ;this is uncool!
```

Previously defined NewTypes can be used within subsequent NewType definitions. The following is an example of a NewType which itself includes another NewType:

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

NewType .object
  position.vector
  speed.vector
  acceleration.vector
End NewType

DefType .object myship       ;see following paragraph!

myship\position\x=100,0,0
```

Note how we now need to use two backslashes to access the fields in myship just like a pathname in DOS.

A NewType, once defined, can be used in combination with both forms of the DefType command just as though it was a another primitive type.

Arrays Inside NewTypes

Besides including primitives and other NewTypes within NewTypes, it is also possible to include arrays inside NewTypes. The square brackets [and] are used when defining arrays inside NewTypes.

Unlike normal arrays, arrays in NewTypes are limited to a single dimension and their size must be dimensioned by a constant not a variable. However the array may be of any type including NewTypes.Also unlike arrays, the dimension size between the square brackets is the size of the array so address.s[4] allocates 4 strings indexed 0...3. The following is an example of using an array inside a newtype:

```
NEWTYPE .record
  name$
  age.w
  address.s[4] ;same as address$[4]
End NEWTYPE

DEFTYPE .record p

p\address[0]=”10 St Kevins Arcade”
p\address[1]=”Karangahape Road”
p\address[2]=”Auckland”
p\address[3]=”New Zealand”
```

```
For i=0 To 3
  NPrint p\address[i]
Next

MouseWait
```

The [index] can be omitted in which case the first item (item 0) will be used.

Defining an array inside a newtype with 0 elements creates a union with the following field (both fields occupy the same memory in the NewType).

The UsePath Directive

Often when using complex NewTypes, path names to access fields within fields can become very long.

Often a routine will be dealing only with one particular field within a NEWTYPE. By using the UsePath directive large path names can be avoided.

When a backslash precedes a variable or field name Blitz2 will insert the UsePath path definition when it compiles the program.

The following code:

```
UsePath shapes(i)\pos

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

is expanded internally by the compiler to read:

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

UsePath can help make routines a lot more readable and can save a lot of typing!

Note: UsePath is a compiler directive, meaning that it affects the compiler as it reads through your program top to bottom not the processor when it runs your program.

This means that if a routine jumps to somewhere else in the program the UsePath in effect will be governed by the closest previous UsePath in the listing.

ARRAYS

Arrays in Blitz2 follow normal BASIC conventions. All Arrays MUST be dimensioned before use, may be of any type (primitive or NewType) and any number of dimensions.

All arrays are indexed from 0...n where n is the size. As with most BASICs an array such as a(50) can actually hold 51 elements indexed 0...50 inclusive.

An array will be of default type unless a .type suffix is added to the array name:

```
Dim a.w(50)      ;fan array of words
```

The ability to use arrays of NewTypes often reduces the number of arrays a BASIC program actually requires.

The following:

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

can be implemented with the following code:

```
NEWTYPE .Alien
  flags.w
  x.w
  y.w
End NEWTYPE
Dim Aliens.Alien(100)
```

You may now access all of the required alien data using just one array. To set up all of the aliens x and y entries with random coordinates

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

This also makes it much easier to expand the amount of information for the aliens simply by adding more entries to the NewType definition, no new arrays required.

Note: unlike most compilers, Blitz2 DOES allow the dimensioning of arrays with a variable number of elements for example: Dim a(n). Also strings in arrays do not require a maximum length setting as is the case with some other languages.

LISTS

Blitz2 also supports an advanced form of the array known as the list. Lists are arrays, but with slightly different characteristics.

Often only a portion of the elements in an array will be used and the programmer will keep a count in a separate variable of how many elements are currently stored in the array. In this situation the array should be replaced with a list which will make things both simpler and faster for managing the array.

Dimming Lists

A list is dimensioned similar to an array except the word List is inserted after the word Dim. Lists are currently limited to only one dimension.

Here is an example of setting up a list:

```
NEWTYPE .Alien
  flags.w:x:y
End NEWTYPE

Dim List Aliens.Alien(100)
```

The difference between a list and an array is that Blitz2 will keep an internal count of how many elements are stored in the list (reset to zero after a Dim List) and an internal pointer to the current element within the list (cleared after a Dim List).

Adding Items to a List

A list starts out as empty, items can be added using the AddItem and AddLast functions. Because the list might be full both commands return a true or false to indicate whether they succeeded.

The following adds one alien to the previously dimmed list:

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

Note how there is no index variable inside the brackets in either use of Aliens(). Although Blitz2 will not flag an error when an index is used, indexes should never be used with list arrays. The empty brackets represent the current item in the list, in this case, the newly added item.

Because AddItem returns false when the list is full we can use a While...Wend loop to fill an entire list with aliens (then kill ‘em off slowly!):

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

The above loops until the list is full. If we wanted to add 20 aliens to a list we could use a For...Next but still need to check if the list was full each time we add an alien:

```
For i=1 To 20
  If AddItem(Aliens())
    Aliens()\x=Rnd(320)
    Aliens()\v=Rnd(200)
  Endif
Next
```

Note that lists can be dimensioned to hold any type not just aliens!

Processing Lists

As mentioned, when an item is successfully added, that item becomes the current item. This current item may then be referenced by specifying the list array name followed by empty brackets ( ).

To process a list (loop through all the items added to a list), we reset the list pointer to the beginning using ResetList and then use the NextItem command to step the pointer through the items in the list. This internal pointer points to the current item.

The following moves all the aliens in the list in a rather ineffective manner (towards the middle of the screen I suspect):

```
USEPATH Aliens()
ResetList Aliens()

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

The While NextItem(Aliens())...Wend structure loops until each item in the list has been the current item. This means that any alien that has been added to the list will be processed by the loop.

The function NextItem returns false if the loop comes to the end of the list.

Again, NextItem returns a true or false depending on whether there actually is a next item to be processed. This example illustrates the convenience lists offer over normal arrays, no "For i=1 To num" to step through arrays using the old index method, instead a clean While...Wend with a system that is faster than normal arrays!

Removing Items From a List

It is often necessary to remove an item from a list while you are processing it. This may be achieved with KillItem. This example again works with the Aliens list:

```
ResetList Aliens()
While NextItem(Aliens())
  If Aliens()\flags=-1 ;if flag=-1
    KillItem Aliens() ;remove item from list
  Endif
Wend
```

Note: After a KillItem, the current item is set to the previous item. This means the While NextItem() loop will not miss an item if an item is removed.

List Structure

Although it is possible to access items in a list by treating them as normal arrays with an index variable it should never be attempted.

The order of items in a list is not always the same as the order they are in memory. Each item contains a pointer to the item before and the item after. When Blitz2 looks for a next item it just looks at the pointer attached to the current item and its physical memory location is NOT important. When an item is added to a list, an arbitrary memory location is used, the current item's NextItem pointer is changed to point to the new item and its old value is given the new items NextItem pointer.

Confused? Well don't worry, just don't ever treat lists as normal arrays by trying to access items with the index method.

The Pointer Type

This is a complex beast in Blitz2. When you define a variable as a pointer type you also state what type it is pointing to. The following defines biggest as a pointer to type Customer.

```
DefType *biggest.Customer
```

The variable biggest is just a long word that holds a memory location where some other Customer variable is located.

For example, we may have a large list of customers. Our routine goes through them one by one and if the turnover of a customer is larger than the one pointed to by biggest then we point biggest towards the current customer:

```
*biggest=CustomerArray()
```

Once we have looped through the list we could print out the biggest data just as if it was type Customer when it is actually only a pointer to a variable with type customer with the following code:

```
Print *biggest\name
```

4. PROCEDURES

Introduction

A procedure is a way of ‘packaging’ routines into self contained parts of the program. Once a routine is packaged into a procedure, it can be called from your main code. Parameters can be passed, and an optional value returned to your main code. Because a procedure contains its own local variable space, you can be sure that none of your main or global variables will be changed by the calling of the procedure. This feature means procedures are very portable, in effect they can be ported to other programs with out conflicting variable name hassles.

Procedures that return a result are called functions in Blitz2, ones that do not return a result are known as statements.

Functions and Statements in Blitz2 have the following characteristics:

- The number of parameters is limited to 6
- Gosubs and Gotos to labels outside a procedure's code is strictly illegal
- Any variables used inside a procedure will be initialised with every call Statement

Statements

A procedure that does not return a value is called a Statement in Blitz2. An example of a statement type procedure which prints the factorial of a number is:

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

For k=1 To5
  fact{k}
Next

MouseWait
```

Use of curly brackets { and } to both define parameters for the procedure, and in calling the procedure. These are required even if the procedure requires no parameters.

If you type in this program, compile and run it, you will see that it prints out the factorials of the numbers from 1 to 5. You may have noticed that the variable k has been used in both the procedure and the main code. This is allowable because the k in the procedure is local to the fact procedure, and is completely separate from the k in the main program. The k in the main program is known as a global variable.

You may use up to six variables to pass parameters to a procedure. If you require more than this, extra parameters may be placed in special shared global variables.

Also, variables used to pass parameters may only be of primitive types, you cannot pass a NewType variable to a procedure however you can pass pointer types.

Functions

In Blitz2, you may also create procedures which return a value, known as functions. The following is the same fact procedure implemented as a function:

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

For k=1 To5
  NPrint fact{k}
Next

MouseWait
```

Note how Function Return is used to return the result of the function. This is much more useful than the previous factorial procedure, as we may use the result in any expression we want. For example:

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

A function may return a result of any of the 6 primitive types. To inform a procedure what type of result you are wanting to return, the type descriptor may be appended to Function command. If this is omitted, current default type will be used (normally .q).

The following is an example of a string function:

```
Function$ spc{n}
  For k=1 To n
    a$=a$+” “
  Next
  Function Return a$
End Function

Print spc{20},”Over Here!”

MouseWait
```

Recursion

The memory used by a procedure's local variables is unique not only to the actual procedure, but to each calling of the procedure. Each time a procedure is called a new block of memory is allocated and freed only when the procedure ends.

The implications of this are that a procedure may call itself without corrupting its own local variables. This allows for a phenomenon known as recursion. Here is another version of the factorial procedure which uses recursion:

```
Function fact{n}
  If n>2 Then n=n*fact{n-1}
  Function Return n
End Function

For n=1 To 5
  NPrint fact{n}
Next

MouseWait
```

This example relies on the concept that the factorial of a number is actually the number multiplied by the factorial of one less than the number.

Accessing Global Variables

Sometimes it is necessary for a procedure to access one or more of a programs global variables. For this purpose, the Shared command allows certain variables inside a procedure to be treated as global variables.

```
Statement example{}
  Shared k
  NPrint k
End Statement

For k=1 To 5
  example{}
Next

MouseWait
```

The Shared command tells Blitz2 that the procedure should use the global variable k instead of creating a local variable k. Try the same program with the Shared removed. Now, the k inside the procedure is a local variable, and will therefore be 0 each time the procedure is called.

Procedures Summary

Blitz2 supports two sorts of procedures, the function and the statement. Both are able to have their own local variables as well as access to global variables through the use of the Shared statement.

Up to six values can be passed to a Blitz2 procedure.

A Blitz2 function can return any primitive type using the Function Return commands.

Using Assembler in Blitz Procedures

Procedures also offer an excellent method of incorporating assembly language routines into Blitz programs.

The Statement or Function is defined as usual with a list of parameters enclosed in curly brackets. When using assembler, the parameters passed to the procedure are loaded in data registers D0-D5.

Care must be taken to ensure that address registers A4-A6 are restored to their initial state before the code exits from the procedure using the AsmExit command.

To set the return value in assembler for Functions simply load the register D0 with the value before the AsmExit command.

For an example of an assembler procedure in Blitz, turn to page 68.

5. BLITZ ERROR CHECKING AND DEBUGGING

Compile Time Errors

Blitz2 reports two types of errors. Compile time errors are those found when Blitz attempts to compile your code and runtime errors occur when the program is being executed.

The first type, compile time errors, cause a message to appear on the editor screen. When OK is selected you are returned to the offending line of code in your program.

Appendix 2 of the Blitz2 Reference Manual contains a description of all the possible errors at compile time. The following list repeats some Blitz2 rules that have to be abided by for your program to be successfully compiled:

1. Any Blitz functions (commands that return a value) must have their parameters contained inside brackets:

```
If ReadFile(0,”ram:test”)
```

2. Blitz commands that are not functions must not have their parameter in brackets:

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

3. Using a .type suffix when referring to items in a NewType will cause garbage at end of line error:

```
person\name$=”Ivan” ;(drop the $)
```

4. A numeric variable can only be one .type, a Mismatched Type error will occur if you attempt to use a different .type suffix further down the program with the same variable name (with the exception of string variables).

Of course there are many hundreds of mistakes that can cause your program to fail to compile, most will require a quick look in the Blitz2 Reference Manual to check the syntax of a command and maybe cross reference your code with one of the examples.

Don't forget the Help key can be used to quickly check the syntax of a command.

The CERR Directive

When using macros and conditional compiling you may wish to generate your own compile time errors.

The CERR directive is used to generate user defined compile time errors. The following will halt the compiler and generate the message, “Should Have 3 Parameters”:

```
CERR “Should Have 3 Parameters!”
```

See conditional compiling in Chapter 9 for more information on CERR.

Runtime Errors

Errors that occur while your program is executing are called runtime errors. When developing programs in Blitz, the Runtime Error Debugger should always be enabled on the Compiler Options. If it's not and an error occurs the system will crash.

If you need to run your program without runtime errors enabled for speed purposes a SetErr directive should be included to stop the system crashing, the system will then jump to the code listed after the SetErr.

The following line included at the top of your program is suggested:

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

Any programs that use file handling should always include some sort of error trapping to handle situations where program cannot locate a file, or the file is of the wrong type.

Any operating system based software should also always include error checking as screens and windows may fail to open due to low memory.

You may also setup an error handler just for one section of code. The SetErr... errorhandler...End SetErr should be at the start of the section and a ClrErr at the end. The following will flash the screen and end if LoadShapes fails:

```
SetErr
  DisplayBeep_ 0
  End
End SetErr

LoadShapes 0,"filename"

ClrErr
```

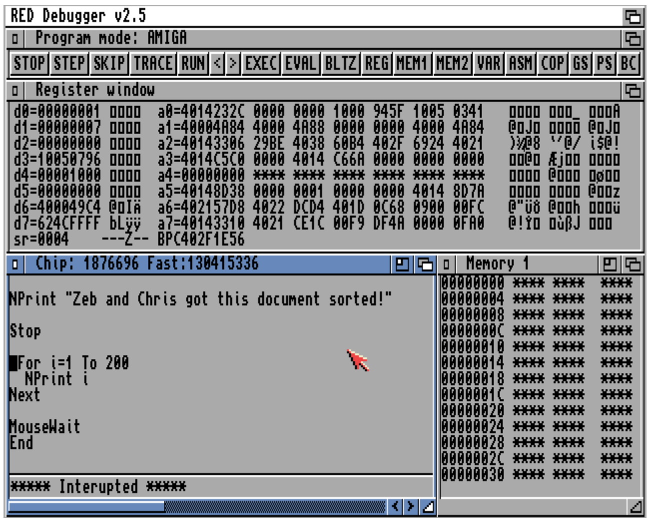
The Blitz Debugger

If a runtime error occurs when the program is run from the editor, the Blitz2 debugger will be activated. Runtime errors must also be enabled in the compiler options requester.

The debugger will not be activated if there is an error-handler already enabled in the program using the SetErr command.

The debugger can also be activated by using the CTRL/ALT C keys, clicking on the “BRK” gadget of the debugger window or including a STOP command in your program.

The debugger is a powerful tool in finding out causes of errors and locating bugs. The ability to step back through code executed prior to the break gives the programmer an excellent understanding of how an error has occurred. The following is a screenshot of the debugger after the program encountered a STOP command.



```
d0=00000001 □□□□  a0=4014232C 0000 0000 1000 945F 1005 0341   □□□□ □□□_ □□□A
d1=00000007 □□□□  a1=40004A84 4000 4A88 0000 0000 4000 4A84   @□J□ □□□□ @□J□
d2=00000000 □□□□  a2=40143306 29BE 4038 60B4 402F 6924 4021   )¾@8 `´@/ i$@!
d3=10050796 □□□□  a3=4014C5C0 0000 4014 C66A 0000 0000 0000   □□@□ Æj□□ □□□□
d4=00001000 □□□□  a4=00000000 **** **** **** **** **** ****   □□□□ @□□□ □ø□□
d5=00000000 □□□□  a5=40148D38 0000 0001 0000 0000 4014 8D7A   □□□□ □□□□ @□□z
d6=400049C4 @□IÄ  a6=402157D8 4022 DCD4 401D 0C68 0900 00FC   @"ÜÔ @□□h □□□ü
d7=624CFFFF bLÿÿ  a7=40143310 4021 CE1C 00F9 DF4A 0000 0FA0   @!Î□ □ùßJ □□□
sr=0004    ---Z-- BPC402F1E56
```

```
NPrint "Zeb and Chris got this document sorted!"

Stop

For i=1 To 200
  NPrint i
Next

MouseWait
End

***** Interupted *****
```

```
00000000 **** ****  ****
00000004 **** ****  ****
00000008 **** ****  ****
0000000C **** ****  ****
00000010 **** ****  ****
00000014 **** ****  ****
00000018 **** ****  ****
0000001C **** ****  ****
00000020 **** ****  ****
00000024 **** ****  ****
00000028 **** ****  ****
0000002C **** ****  ****
00000030 **** ****  ****
```

Note, by making the debugger window larger more of the program can be viewed.

The Debugger Gadgets

The following is a description of the debugger gadgets:

BRK	Click on this to stop a program running and enable the Blitz debugger.
STP	Use this to stop a program during Trace mode.
SKP	Skip causes the debugger to skip a command, program execution will continue from the next command.
TRC	Trace mode allows the programmer to single step through their code, by increasing the size of the debugger window program flow can be viewed.
RUN	Causes program execution to resume after being stopped.
<<	View previous command history allows the programmer to review the commands that were executed prior to the program being stopped.
>>	View forward allows the user to forward through the command history after using the view previous gadget.
EXC	Execute allows the programmer to manually enter a Blitz command to be executed by the debugger.
EVL	Evaluate allows the programmer to view any variable simply by entering its name after clicking on EVL.

Tracing Program Execution

The debugger allows the user to single step through or trace program execution, displaying in its window which command is currently being executed.

STP is used to single step through your program, each time you click on STP the debugger will execute the command pointed to by the arrow and stop. Trace steps continuously through the code displaying each command as it goes. To stop the Trace you must use the STP gadget.

Level is used to change the trace level, if Level is ON, the debugger will not trace or single step through the inside For...Next loops but execute normally until loop exits.

It will also not trace the execution of any procedures or subroutines called, this is most useful for watching the program's main loop while not having to sit through the trace of each subroutine when called.

Resuming Normal Execution

Program execution return normally after debugger is activated using Run gadget.

If the debugger was activated using the STOP command the arrow will be pointing to STOP. Before continuing, the command must be skipped over using the Ignore command. This is true for any command that has caused a runtime error and invoked the debugger.

To return to the editor from the debugger either hit the Escape key or click on the close window gadget of the debugger window.

Viewing Command History

The debugger keeps a record of the commands executed before the program is stopped in a large buffer.

The Back-up command will step backwards from where the program halted, allowing the programmer to view the previous commands executed by the computer. A hollow arrow marks the current position in the history buffer.

Forward command is used to step forwards through the history buffer, attempting to step past where the program was stopped will produce a AT END OF BUFFER error.

These features are invaluable to following through program execution up to where the program was halted. If a program halted in the middle of a subroutine or procedure you can step backwards to find where the routine was called from.

Direct Mode

While the debugger is activated the programmer has two tools available to examine the internal state of the program.

To find out the value of any variables the EVaLuate command can be used. A prompt will appear, after typing the name of the variable and hitting return the value will be printed on the debugger display.

The EXeCute command is used to run a Blitz2 command. A prompt will appear and the programmer can then type in any Blitz2 command such as CLS or n=20.

Debugger Errors

The following errors may occur when using direct mode commands Evaluate & eXecute:

Can’t Create in Direct Mode

Occurs if you try and Evaluate a non existent variable (not created) in the program.

Library Not Available in Direct Mode

This occurs when a Blitz2 command is executed and is from a command library not used by the program. If, for instance, the program doesn't use strings, the string command library will not be part of the object code and so any string type commands will not be able to be executed and generate this error.

Not Enough Room in Direct Mode Buffer

This error should never occur. If it does the object butter size in the Compiler Options requester should be increased.

AT END OF BUFFER

Occurs if the programmer tries to view forward of where the program stopped.

6. BLITZ OBJECTS

This chapter covers the use and handling of Blitz2 objects, structures designed to control multiple system elements such as graphics, files, screens, etc.

Blitz2 looks after all memory requirements for objects including freeing it up when the program ends.

Although most objects have their own specific commands, the standard way they are handled in Blitz2 means the programmer is never faced with unusual syntax. Instead they can depend on a standard modular way of programming the multitude of elements available in Blitz2.

The following is a list of the main Blitz2 objects:

files	for sequential and random access DOS file handling
modules	SoundTracker compatible music objects
blitzfonts	8x8 fonts for fast bitmap text output
intuifonts	any size fonts for window text output
shapes	standard Blitz2 graphics element
palettes	colour palette structure
bitmaps	standard Blitz2 display element
sounds	digitised sound sample element
sprites	Blitz mode hardware sprite element
screens	standard Intuition type screens
windows	standard Intuition type windows
gadgets	standard Intuition type gadgets
menus	standard Intuition type menus

Object Similarities

Blitz2 objects all have a set of commands allowing the program to create or define them, manipulate and of course destroy them.

Most objects have a chapter in the Blitz2 reference manual devoted to them, outlining all the special commands used to create and manipulate the object. All Blitz2 objects can be destroyed using the Free command. If an object has not been destroyed when a program ends, Blitz2 will automatically Free that object.

Free BitMap 0 will free up all memory allocated for object bitmap 0, this is useful when using objects temporarily and will need memory later in the program, otherwise it's usual to let Blitz free up all objects automatically when program ends.

Object Maximums

Each object has its own maximum. This number defines how many of one type of object can be created and manipulated by the program. The maximum can be changed for each object in the Compiler Options window of the editor.

The runtime error Value Out of Maximum Range means you have tried to use an object number greater than that set in the maximums window of Compiler Options.

Using an Object

Many commands need previously created objects present to operate properly. For example, the Blit command, which is used to place a shape onto a bitmap, needs both a previously created shape object and a bitmap object.

When you use the Blit command, you specify the shape object to be blitted and Blitz will blit that shape onto the currently used bitmap.

```
Use BitMap 0        ;make bitmap the currently used bitmap
Blit 3,10,10        ;blit shape 3 onto currently used bitmap
```

The Use command in the previous example makes BitMap 0 the currently used bitmap. Screens, Windows and Palettes are three other Blitz2 objects that often need to be currently used, for commands to work properly.

Note, when an object is created, it becomes the currently used object of its class.

Blitz2 makes extensive use of this currently object idea. Its advantages include faster program execution, less complex commands and greater program modularity.

Input/Output Objects

Bitmap, file and window objects can all operate as I/O devices. ObjectInput and ObjectOutput commands allow the user to channel input and output to different places.

The Print command will always write to the current output object, edit and inkey$ will always attempt to read from the current input object.

```
WindowOutput 2      ;window 2 is the current output object
Print “HELLO”
BitMapInput 1       ;make bitmap 1 the current input object
a$=Edit$(80)
```

Object Structures (for advanced users)

Appendix 1 of the Blitz2 reference manual contains descriptions of each of the Blitz2 object's structures. The Addr command is used to find the location in memory the structure of a particular object.

Advanced users can use the Addr command with Peek & Poke and inline assembler routines to access important values in an object's structure. This is often helpful with system type objects such as Screens and Windows that contain pointers to their Intuition counterparts.

The following calls the system command ScreenToFront_ obtaining the location of the Intuition Screen structure from the Blitz2 Screen object in memory.

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

The next listing illustrates obtaining a window's system structure and assigning it to a pointer type .window. AmigaLibs.Res should be resident before run this example.

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

Note: the NewType .Window refers to the system (Intuition) window structure where as the NewType .window refers to the Blitz2 window structure.

Overview of the Primary Blitz2 Objects

Screens

These are created using Screen and FindScreen commands. The first will open a new screen while the second will make an existing screen (usually Workbench screen) a Blitz2 screen.

Free Screen n should only be attempted after any windows open on the screen are closed (free'd) first.

Screen objects both configure the resolution of the display and its palette as well as being the place where windows are opened. Any windows opened, RGB or UsePalette commands will use the currently used screen.

The function Peek.l(Addr Screen(n)) can be used to obtain the location of the system .Screen structure when calling system routines.

Windows

Created with the Window command. Gadgets and menus are always added to the currently used window while the drawing commands WPlot, WCircle, WLine and WBox all render to the currently used window.

Window objects can be used for input/output using WindowInput and WindowOuput commands. The cursor position for in/out can be controlled with WLocate command. Windows can be freed without worry of freeing any attached gadget or menulists.

Gadget and Menu Lists

Gadgets and menus must be grouped together in Blitz objects known as yes, you guessed it, gadgetlists and menulists. These lists are attached to a window when the window is first created (opened). This means that gadgets and menus should all be pre-defined in their lists at the start of the program.

Palettes

A palette object contains RGB information for each of the colours in a display. Palettes are a little different to regular Blitz objects in the following ways.

Use Palette will set the current screen or slice to the colours in the palette.

The RGB command as well as the Red(), Green() and Blue() functions apply to the colours in the current slice or screen NOT in the current palette.

There is no create palette command, they are either created when loaded from an IFF file or when using PalRGB. If no palette object exists with either command Blitz2 will create one for you.

BitMaps

A bitmap refers to the array of pixels that make up the display. A bitmap can either be created with the BitMap command, loaded from disk or fetched from a screen using the ScreensBitMap command.

A bitmap command can be freed using the Free BitMap *n* command, you can not free bitmaps created with the ScreensBitMap command.

As with windows, bitmaps can be used as input/output devices with BitMapInput and BitMapOutput commands. These are used primarily in BlitzMode.

In BlitzMode the keyboard should be enabled with BlitzKeys On before attempting to use BitMapInput.

When using BitMapOutput the Locate command can be used to position the cursor.

Shapes

Shapes are used to contain graphic images. They can be initialised by either loading them from disk or being clipped from a bitmap object using GetAShape command.

Shapes are freed using standard Free Shape *n* syntax. Shapes should not be freed if they are used with gadgets or menu items until relevant gadget or menulist is freed 1st.

There are many powerful commands to manipulate shapes, including rotation and scaling.

Sprites

Sprites are initialised by either loading them from disk or converting a shape object to a sprite object using GetaSprite. The shape object can be freed once it has been converted to a sprite.

```
Free Sprite n     ;will free a sprite
```

Sprites can currently only be used in Blitz mode however in Amiga mode, the pointer can be assigned to a single sprite object.

Slices

A slice is used to configure a display in Blitz mode. They are initialised with Slice command.

Unlike other objects, single slices cannot be freed. FreeSlices is used to free all slices currently initialised.

The commands Show, ShowF, ShowB and ShowSprite all use the currently use slice.

The RGB command also affects the colour registers in the currently used slice as does the Use Palette command.

Files

Unlike other Blitz objects files are opened and closed rather than initialised & killed.

Files are initialised with OpenFile(), ReadFile() and WriteFile() functions. Unlike other Blitz objects a function is used so the program can tell if file was opened.

CloseFile n command is used to ‘free’ a file object. The command Free File n may also be used, unlike other objects it is best to close all files yourself rather than rely on Blitz2 to close them when the program exits.

A file is of course an input/output object, the commands FileInput and FileOutput are used to direct input and output to files.

Get, Put, ReadMem and WriteMem require file# parameters and so do not require the use of FileInput and FileOutput commands.

Objects Summary

Blitz2's objects are custom data structures used by the libraries to handle a whole assortment of entities. Blitz2 manages the memory required of these structures, freeing them automatically when a program ends.

They provide a simple interface to many of the more complex Blitz2 commands. Parameter passing is minimised as many of the commands take advantage of the currently used object.

As libraries are upgraded and added to Blitz2, more objects will be added and versatility and functionality of existing objects will be increased.

7. BLITZ MODE

Although the Amiga's operating system is very powerful, its ability to take full advantage of the graphics capacity of the machine is limited. Blitz mode is for programmers wanting to produce smooth animated graphics for games and the like.

The command Blitz puts your program in Blitz mode. When this happens the operating system is disabled and your program takes over the whole machine. This means that it will not multi-task and file access is no longer possible.

The benefits of Blitz mode are that programs run a lot quicker and display options such as smooth scrolling and dual-playfield are possible. Blitz mode is not a permanent state. When your program re-enters Amiga mode or exits, the operating system is brought back to life as though nothing happened.

Careful attention must be paid regarding entering Blitz mode as version 1.3 and older of the operating system can take up to 2 seconds to flush any buffers after a file is closed. You should always ensure that absolutely no disk or file access is taking place before entering Blitz mode. At the time of this writing, no software method of achieving this has yet been discovered. The best we can suggest is that a VWAIT 100 should always be executed before using Blitz mode.

Slice Magic

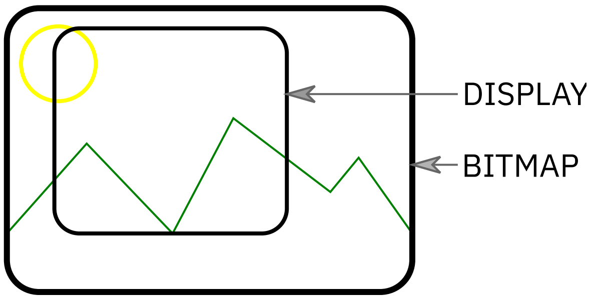
The designers of the Amiga hardware have implemented many features for achieving smooth, fast graphics. After entering Blitz mode the display is controlled using Slices. Slices are much more flexible than the operating system's screens, they allow features such as smooth scrolling, double buffered displays and much more.

The ability to have more than one slice means that the display can be split into different regions each with their own resolution.

The following is a description of the main display features accessible with slices:

Smooth Scrolling

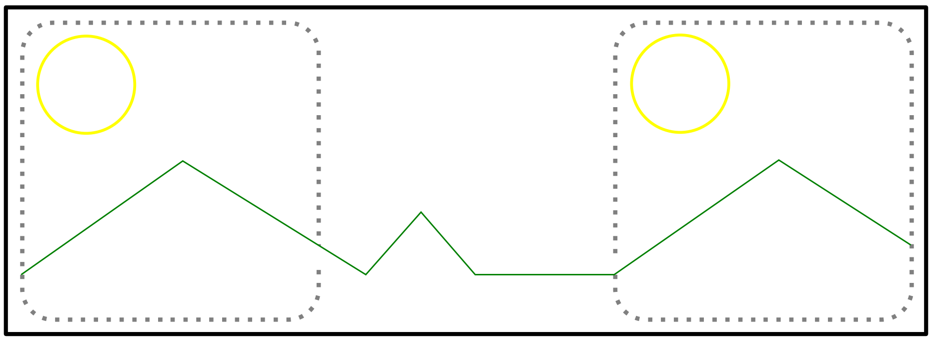
This is achieved by displaying only a portion of a large bitmap. The Amiga hardware enables us to move the display window around the inside of a large bitmap as the following diagram shows:



The display window represents what is shown on the monitor, as we move the display window across the bitmap to the right the image we see on the screen scrolls smoothly to the left.

The Blitz commands Show, ShowF and ShowB allow us to set the position of the display window inside the bitmap.

The above diagram limits the amount we can scroll to the size of the bitmap. By duplicating the left portion of the bitmap on the right we can smoothly scroll the display across. When it reaches the right, reset it back to the far left. As there is no change when the display is reset to the left the illusion of continuous scrolling is created.

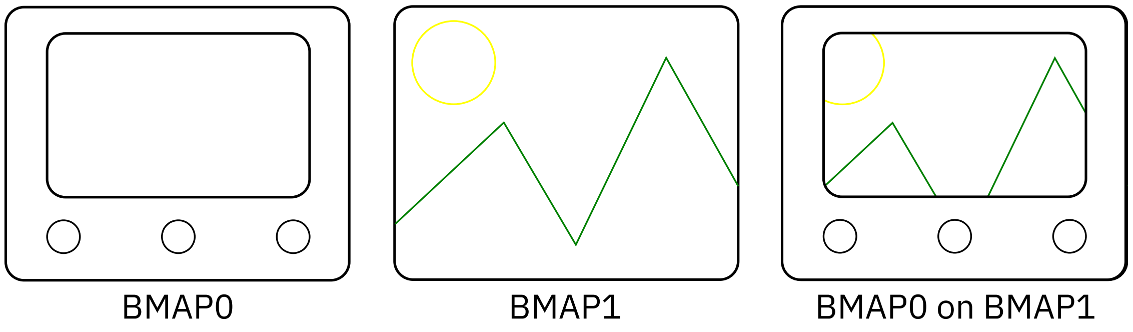


The above left right scenario also applies to vertical scrolling (up and down).

Dual-Playfield

In some situations, the display will be made up of a background and a foreground. The Amiga has the ability to display one bitmap on top of the other called dual-playfield mode to achieve this effect.

In a dual playfield display, two 8 colour bitmaps can be displayed, one in front of the other and any pixels set to colour zero in the front playfield will be transparent letting the back playfield show through. Each playfield can have its own colours.



Copper Control

Smooth animation is achieved by moving graphics in sync with the video display. The display is created by a video beam that redraws the screen line by line every 50th of a second. Often, it's useful to sync things to the vertical position of the vertical beam. This is achieved using the Amiga graphics co-processor known as the Copper.

Blitz2 offers several ways of taking advantage of the copper hardware. The most popular is to change the colour of the background colour to produce rainbow type effects on the display. This is achieved using the ColSplit command.

Those with good knowledge of Amiga hardware may wish to program copper to make other changes at different vertical places, this can be achieved using CustomCop.

The Blitter

The Amiga has custom hardware specifically to transfer graphic images onto bitmaps known as the blitter. Blitz2 offers several ways of blitting shapes onto a bitmap and a special Scroll command to shift areas of a bitmap around, also using the blitter.

The following is a brief overview of the various blitter based commands in Blitz2:

Blit	Put shapes onto bitmaps.
QBlit	Same as Blit but Blitz2 remembers where the shape was put and will erase it when it's time to move the shape somewhere else on the bitmap.
BBlit	Same as QBlit but when it is time to move the shape, instead of erasing the shape, Blitz2 replaces what was on the bitmap previous to the BBlit.
SBlit	Sames as Blit but with a stencil feature which protects certain areas of the bitmap from being blitted on.
Block	Fast version of Blit that works only with rectangular shapes a multiple of 16 pixels wide.
ClipBlit	Slow version of Blit which will clip the shape to fit inside the bitmap.
Scroll	Used to copy sections of a bitmap from one position to another.

QAmiga Mode

It's also possible to jump out of Blitz mode and back into Amiga mode. This can be done using either the QAmiga or Amiga statement.

Using Amiga to go back into Amiga mode will fully return you to the Amiga's normal display, complete with mouse pointer.

Using QAmiga will return you to Amiga mode, but will not affect the display at all. This allows Blitz mode programs to jump into Amiga mode for operations such as file I/O, then jump back to Blitz mode without having to destroy Blitz mode display.

An Important note!!!!!

You should always ensure that absolutely no disk or file access is taking place before entering Blitz mode. At the time of this writing, no software method of achieving this has yet been discovered.

By following these guidelines using Blitz mode should be pretty safe:

- Always wait for the floppy drive light to go out if you have saved some source code before Compiling/Running a program which launches straight into Blitz mode.
- A590 hard drive users - always wait for the second blink of the drive light when using Workbench 1.3. Workbench 2.0 users have their buffers flushed in one go.
- If you use the QAmiga statement for the purpose of writing data to disk, its a good idea to execute a delay before go back to Blitz mode - in effect, simulating the above. Executing a VWait 250 will provide a delay of about 5 seconds - a safe delay to use. After reading data use a VWait 50. Another important thing to remember about Blitz mode is that any commands requiring the presence of the OS become unavailable while you're in Blitz mode. If you attempt to open a window in Blitz mode, you will be greeted with an ‘Only available in Amiga Mode' error at compile time. For this reason, the Reference Guide clearly states which commands are available in which mode.

The Blitz, Amiga, and QAmiga statements are all compiler directives. This means they must appear in the applicable order in your source code.

Summary

Blitz2 provides two environments for your programs to execute in. Amiga mode should be used for any applications software and whenever your game needs to load data from disk. Blitz mode is for programs that need to take advantage of the special display modes we have provided in Blitz2. These provide performance that is just not available in Amiga mode but will halt the Amiga's operating system.

To conclude, the only time it is acceptable to close down the Amiga's multi-tasking environment is when the software is dedicated to entertainment. Any applications software that uses Blitz mode will NOT be welcomed by the Amiga community.

Resident Files

To make writing programs which manipulate large number of NewTypes, macros or constants easier, Blitz2 includes a feature known as resident files.

A resident file contains a pre-compiled list of NewTypes, macros and constants. By creating resident files, all these definitions can be dropped from the main code making it smaller and faster to compile.

To create a resident file you will need a program which contains all the NewTypes. macros and constants you want to convert to resident file format.

The following is an example of a such a program:

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

Macro mac
  NPrint "Hello"
End Macro

xconst=10
```

To convert these definitions to a resident file, all you need to do is COMPILE & RUN the program, then select CREATE RESIDENT from the COMPILER menu. At this point, you will be presented with a file requester into which you enter the name of the resident file you wish to create. That's all there is to creating a resident file!

Once created, a resident file may be installed in any program simply by entering the name of the resident file into one of the 'RESIDENT' fields of the compiler options requester. Once this is done, all NewType, macro and constant definitions contained in the resident file will automatically be available.

Resident file AMIGALIBS.RES contains all the structures, constants and macros associated with the Amiga OS. Those familiar with programming the OS will not have to include all the usual library header files and will save minutes every compile time.

Operating System Calls

Much effort has been made to let the Blitz2 programmer make the most of the Amiga's powerful operating system.

Calling Operating System Libraries

Often the programmer with a good knowledge of the OS will want to access routines that have not been supported by the internal Blitz2 command set. All routines in the Exec, Intuition, DOS and Graphics libraries are accessible from Blitz2.

Support for other Amiga standard libraries is available by purchasing the Blitz2 advanced programmers pack from Acid Software.

The following is an example of call routines in Amiga ROMs graphics & intuition libraries:

```
FindScreen 0                                    ;use Workbench screen
;open gimmezerozero window

Window 0,0,10,320,180,$408,””,1,2
rp.l=RastPort(0)                                ;get rastport for window
win.l=Peek.l(Addr Window(0))                    ;find window structure

DrawEllipse_ rp,100,100,-50,50                  ;graphics library
MoveWindow_ win,8,0                             ;intuition library
BitMap 1,320,200,2                              ;setup work bitmap
Circlef 160,100,100,1                           ;draw something

;then transfer it to window

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

WaitEvent
```

The final command BltBitMapRastPort_ is very useful for transferring graphics drawn with the faster bitmap based Blitz2 commands onto a Window. This is a very system friendly way of achieving this objective.

Accessing Operating System Structures

With the file AMIGALIBS.RES resident (see the start of this chapter) even more control of the OS is possible. The following is an example of accessing OS structures.

```
;variable *exec points to the ExecBase struct
;variable *mylist points to a List type
;variable *mynode points to a system node

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

While *mvnode\ln_Succ
  a$=Peek$(*mynode\ln_Name)      ;print node name
  NPrint a$
  *mvnode=*mvnode\ln_Succ        ;go to next node
Wend

MouseWait
```

The use of the asterisk in *variablename.type means that instead of Blitz2 creating a variable of a certain type it actually just creates a ‘pointer' to that type. The type (structure) can then be accessed just like it was an internal Blitz2 variable.

The command Peek$ is an excellent way of retrieving text from OS structures as it reads memory directly into a Blitz2 string variable until it hits a null (Chr$(0)).

Locating Variables and Labels in Memory

The ampersand (&) character can be used to find the address of a variable in the Amiga's memory. For example:

```
; An example of using ‘&’ to find the address of a var.

Var.l=5
Poke.l &Var,10
NPrint Var
MouseWait
```

This is similar to the VarPtr function supplied in other BASICs.

When asking for the address of a string variable, the returned value will point to the first character of the string. The length of the string is a 4 byte value, located at the address-4.

The ‘?’ character can be used to find the address of a program label in the Amiga’s memory. For example:

```
;An example of finding the address of a program label

MOVE #10,There                  ;wo! assembly code on this line
NPrint Peek.w(?There)
MouseWait
End

There:
Dc.w 0                    ;wo! and again here
```

These features are really only of use to programmers with some assembly language experience who need unconventional means for their ends.

Constants

A constant, in BASIC programming terms, is a value which does not change throughout the execution of a program. The 5 in a=5 is a constant.

A hash sign (#) before a variable name means that it is a constant (no longer a variable!) and cannot change in value when the program is running. The following line means that #width is a constant and will always be 320:

```
#width=320
```

Constants have the following properties:

- Are faster than variables and do not require any memory
- Make programs more readable than using numbers
- Can be used in assembler
- Can be used with conditional compiling evaluations
- Can only hold integer values
- Make it easier to change a constant amount used throughout a program
- Can be altered through the source at compile time but NOT at runtime

The most important aspect of constants from a BASIC programmers point of view is that any ‘magic numbers' that appear throughout their code can be replaced by meaningful words such as #width.

If the program ever has to be modified to work with a new width, instead of going through all the source changing any mention of the number 320, the programmer can just change the constant equate at the top of the program #width=320 to #width=640 etc.

Conditional Compiling

Allows the programmer to switch the compiler on and off as it reads through the source code, controlling which parts of the program are compiled and which are not. Conditional compiling is useful for producing different versions of the same software without using 2 different source codes. It can also be used to cripple a demo version of the software or produce different programs for different hardware configurations.

Tracking down bugs can also involve the use of conditional compiling, by turning off any unnecessary parts of code it becomes easier to pinpoint where exactly the error is occurring. However, we hope the Blitz2 debugger will make this practice obsolete.

The conditional compiler directives are as follows:

CNIF	Compiler on if numeric comparison is true, off otherwise
CSIF	Compiler on if string comparison is true, off otherwise
CELSE	Switch compiler from previous state on=>off off=>on
CEND	End of conditional block (restores previous state)

The compiler has an internal on/off switch, after a CNIF or CSIF comparison the compiler switches on for true, off for false. A CELSE will toggle the compiler switch and the CEND will restore the on/off state to that of the previous CNIF/CSIF.

CNIF/CEND blocks can be nested.

It's important remember that CNIF directive only works with constant parameters – for example, ‘5’, ‘#test’ - and not with variables. This is because Blitz2 must be able to evaluate the comparison when it is actually compiling, and variables are not determined until a program is actually run.

The following code illustrates using conditional compiling:

```
#crippled=1 ;is a crippled version
NPrint “Goo Goo Software (c)1993”
CNIF #crippled=1
  NPrint “DEMONSTRATION VERSION”
CELSE
  NPrint “REGISTERED VERSION”
CEND

; and later on in the program...

.SaveRoutine
CNIF #crippled=0     ;only if not crippled

;do save routine

CEND
Return
```

The benefit over using a straight If crippled=0...EndIf is that the crippled version of the above code will not contain the save routine in the object code so that there is no way it can be uncrippled by hackers.

Conditional compiler directives come into their own when doing macro programming.

Macros

This is a feature usually only found in assemblers or lower level programming languages. They are used to save typing, to replace simple procedures with faster ‘inline' versions, or at their most powerful to generate code that would be impractical to represent with normal code.

A macro is defined in a Macro name...End Macro structure. The code between these two commands is not compiled but placed in the compiler's memory. When the compiler reaches a !macroname it then inserts the code defined in the macro at this point of the source code.

The following code:

```
Macro mymacro
  a=a+1
  NPrint “Good Luck”
End Macro

NPrint “Silly Example v1.0”
!mymacro
!mymacro
MouseWait
```

is expanded internally by the the compiler to read:

```
NPrint “Silly Example v1.0”
a=a+1
NPrint “Good Luck”
a=a+1
NPrint “Good Luck”
MouseWait
```

Macro Parameters

To make things a little more useful, parameters can be passed in a macro call using braces, { and }. These parameters are firstly inserted into the macro text, then the macro text is inserted into the main code.

When a macro is defined the use of the back apostrophe (above TAB key on Amiga) before a digit or letter (1-9, a-z) marks the point where a parameter will be inserted.

The following illustrates passing two parameters to a macro:

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

NPrint !distance{20,30}

MouseWait
```

the compiler expands the NPrint line to read:

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

replacing every ‘1 with the first parameter and ‘2 with the second etc.

If there are more than 9 parameters, letters are used; a signifying the tenth parameter, b the eleventh and so-on.

Parameters can be anything, the {20,30} could just as easily been {x,y} in the previous example.

Note: when passing complex expressions as parameters care should be taken to make sure parenthesis are correct:

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

will expand to

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

The above does not expand correctly for the first half. Due to the parenthesis around the second parameter the second half does expand properly.

The ‘O Parameter

This parameter is special as it returns the number of parameters passed to a macro. This is useful for both checking to see that the correct number of parameters was passed as well as generating macros that can handle different numbers of parameters. The following checks to see if two parameters were passed and generates a compile time error if not:

```
Macro Vadd
CNIF ‘0=2
  ‘1=’1+’2
CELSE
  CERR “Illegal number of ‘!Vadd’ Parameters”
CEND
End Macro
!Vadd{a}
```

If you compile & run this program, you will see that it generates an appropriate error message when !Vadd{a} is encountered. The CERR compiler directive is a special directive used to generate a custom error message when a program is compiled.

Recursive Macros

Macros are recursive and can call themselves, the following macro prints the first parameter and then calls itself, minus the first parameter, effectively stepping through the list of parameters passed until a null character (no parameter) reached.

```
Macro dolist ;list upto 16 variables
  NPrint ‘1
  CSIF “’2”>””
  !dolist {‘2,’3,’4,’5,’6,’7,’8,’9,’a,’b,’c,’d,’e,’f,’g}
  CEND
End Macro
!dolist {a,b,c,d,e,f,g,h,i}
MouseWait
```

Replacing Functions with Macros

Macros are an excellent replacement for functions that don’t use any local variables but need to generate more than one return variable. The following macro project takes x, y, z coordinates and projects them onto a 2D x,y plane. It can then be used to generate x,y projections for drawing.

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

#xm=320:#ym=256

Screen 0,28:ScreensBitMap 0,0

For z=-1 5 To 15
  For y=-15 To 15
    For x=-15 To 15
      Circlef !project{x,y,z},3,x&y&z
    Next
  Next
Next

MouseWait
```

The CMake Character

A special character known as the cmake character can be used to evaluate constant expressions and insert the literal result into your code. This can be very useful for generating label and variable names when a combination of macro parameters and constant settings are needed to generate the right label.

```
varR=20
varB=30

Macro Ivar
  NPrint var~’1~
End Macro

!Ivar(2+1)

MouseWait
```

The above example without the cmake characters~ would print 21 as Blitz would expand the code after NPrint to read var2+1, instead it evaluates the expression between cmake characters and generates 3 which it then inserts into the macro text.

Inline Assembler

It's possible to include 68000 machine code inside Blitz2 programs using the inline assembler. This offers the experienced programmer a way of speeding up their programs by replacing certain routines with faster machine code equivalents.

There are three methods of including assembler in Blitz2:

- Inline using the GetReg and PutReg commands to access variables
- Inside statements and functions
- Developing custom Blitz2 libraries

GetReg & PutReg

These commands allow an assembly programmer access to the BASIC variables in the program. The following listing illustrates their use:

```
a.w=5                   ;use words
b.w=10
GetReg D0,a             ;value of a=>D0
GetReg D1,b             ;value of b=>D1
MULU D0,D1
PutReg D1,c.w           ; value of D1=>c
NPrint c
MouseWait
```

The next example inverts first bitplane of bitmap 0. Note how any complex expression can be used after a GetReg command. Because GetReg can only use data registers, we place the location of the bitmap structure in D0 and then move it to A0.

```
Screen 0,3
ScreensBitMap 0,0

While Joyb(0)=0
  VWait 15
  Gosub inverse
Wend

End

inverse:                     ;memory location of bitmap struct=>D0
  GetReg D0,Addr BitMap(0)
  MOVE.l D0,A0
  MOVEM (A0),D0-D1
  MULU D0,D1
  LSR 1#2,D1
  SUBQ #1,D1
  MOVE.l d(A0),A0
loop:
  NOT.l (A0)+
  DBRA D1,loop
Return
```

Using Assembler with Procedures

A more efficient method of using assembler in Blitz2 is to put machine code routines inside functions and statements. Parameters are automatically placed in D0-D5 and if using functions, the value in register D0 will be returned to the calling routine.

The following listing illustrates the use of assembler in the statement qplot{} which sets a pixel on the first bitplane of the bitmap supplied.

Note how more than one assembly instruction can be used per line of source code.

```
Statement qplot{bmap.l,x.w,y.w}
  MOVE.l D0,A0:MULU (A0),D2
  MOVE.l d(A0),A0:ADD.l D2,A0
  MOVE D1,D2:LSR#3,D2:ADD D2,A0:BSET.b D1,(A0)
  AsmExit
End Statement

Screen 0,1.0000000000000000
ScreensBitMap 0,0
bp.l=Addr BitMap(0)

For y.w=0 To 199
  For x.w=0 To 319
    qplot{bp,x,y}
  Next
Next

MouseWait
```

Programmers wanting to develop their own libraries of machine code routines should purchase the Blitz2 advanced programmer’s pack from Acid Software. Blitz2 contains a powerful library system giving the experienced machine code programmer a highly productive and powerful environment to develop advanced software.

9. PROGRAMMING TECHNIQUE & OPTIMISING

Label and Variable Names

The following are rules to adhere to when using variable and label names in Blitz2.

- Names can be of any length
- They must start with a letter (a...z, A...Z) or an underscore
- Must only contain alphanumeric chars and underscores
- Must not start with the same letters as any Blitz2 command

Also, label and variable names in Blitz2 are always treated as case-sensitive, this means that the variables myship and MyShip are entirely different.

Style

There are many variable and label naming approaches that can make programming much easier. The following are a few guidelines that can help keep things in control as your program grows in size and more and more variables and labels are in use. Consistency is essential, if you use any of the following styles, stick to them.

By separating different groups of variables and labels with the following methods, names can have added meaning.

- Full caps “NAME”, initial cap “Name” and lower case “name”
- Letters “l”, words “Loop” and double words “MainLoop”
- Initial underscore “_loop” and mid underscores “main_loop”
- Numeric suffixes such as “loop!”, ”loop2” etc.

Nomenclature is a personal thing and by sticking to a certain style with variable and label names many debugging problems can be avoided. Using good names for everything can make your program far more readable and will greatly aid in finding mistakes.

Common Naming Related Problems

The following is a summary of certain problems that can arise when variable and label names become messy.

- Using the wrong variable name will often not flag an error. If it has not previously been assigned, Blitz2 will create a new variable with a default value of zero. Avoiding a mix of different naming styles will greatly reduce these mistakes.
- Forgetting variable names can slow program development. By using logical names and keeping a list of your main variables on a scrap of paper next to your keyboard helps keep things organised.

- Using lengthy names can aid readability, however it will also increase incidents of typing errors and slow development.
- Use of rude or obscene labels can make programming a little more enjoyable, however it should be avoided if your source code will be read by others.

Remarks and Comments

Other BASICs use REM statement but Blitz2 uses the semicolon (;) character. Anything after a semicolon on a line will be ignored by the compiler. This feature is used to document programs.

Adding remarks, the programmer can document each routine in a program for future reference. One of the main curses of programming is having to return to a section of code developed earlier only to find you can not make head or tail of its logic.

Although it can seem a little tedious, adding accurate explanations of each routine as you write it will save many headaches later.

A section of documentation at the top of programs is also useful. Version number, copyright information, lists of bugs fixed and when as well as full descriptions of all main variables should all be maintained at the top of your program.

Structured Programming Techniques

One main technique in developing structured programs is a method known as indenting. Indenting means that instead of each line being flush with the left margin, spacing is inserted at the start of the line to indent it across the page.

Indenting lines of code that are ‘nested' inside loops or other program flow structures creates a useful aid in visualising the structure of your source code.

The Blitz2 editor has features for indenting code. Tab key is used to move cursor across the page. By changing the tab setting in Ted, the default size of indents can be altered.

By highlighting a block of code, block tab and untab (Amiga [and Amiga]) will move the whole block left or right. Shift cursor left will move the cursor to the same indent as the line above.

Keeping Things Modular

There is nothing more valuable than good initial planning when it comes to developing software. Breaking down your project into modular pieces before you start is a must to avoid the creation of huge spaghetti nightmares.

After deciding on how each section of the program is going to function it is usually best to start with the most difficult sections. Getting the hardest bits going first while the program is small can save a lot of headaches in the long run.

Time spent waiting for your program to compile & initialise compounds itself when you're bug hunting or making small adjustments to certain sections of code. In these situations it is usually best to remove the code from the main program, spend an hour writing a shell that you can test it in and then set about making it bullet proof.

A few things to keep in mind when developing routines:

- Make sure it will handle all possible situations called for
- Convince yourself you are using the most efficient method
- Keep it modular; ie. the routine must return to where it was called
- Keep it well documented
- Include comments regarding global variables and arrays it uses
- Make sure it's bullet proof (won't fall over with bad parameters)
- Indent nested code and limit lengths of lines to aid readability

Along the way...

Besides keeping routines well documented it's always a good idea keep a piece of paper handy to write down notes on the important bits. Lists of variables that are common between routines as well as things ‘to do' in unfinished routines should always be noted.

The ‘to do' list is always a good way of thinking out all problems in advance. Always keep in mind what extra routines will be needed to implement the next one on the list.

One of the biggest mistakes a programmer can make is start a routine that needs all sorts of other routines to function. By starting with the stand-alone/independent bits you can make sure they are working. This keeps you well clear of the headaches caused where you have just added 5 routines, tested none of them and are trying to find a bug which could be located in any of them. Developing a modular approach to programming is definitely the most effective way of finishing a piece of software.

Keeping Your Code Readable

This is next on the list of requirements that will aid in the completion of a piece of software.

The two main keys to readability are indenting nested code and keeping the amount of code on one line to a minimum. The following is an illustration of indenting nested code:

```
If ReadFile(0,”phonebook.data”)
  Filelnput 0
    While Not Eof(0)
      If AddItem(people())
        For i=0 To #num-1
        \info[i]=Edit$(128)
      Next
    Endif
  Wend
Endif
```

This method means that it's easy to see at a glance what code is being executed inside each structure. Using this method it's difficult to make a mistake like leaving out the terminating EndIf or Wend as just by finding the line above at the same level of indentation we can match up each Wend with its corresponding While etc.

Optimising Code

It's always important to have a firm grasp on how much time is being taken by certain routines to do certain things. The following are a few things to keep in mind when trying to get the best performance from your Blitz2 programs.

Performance is most important with arcade type games where a sluggish program will invariably destroy the playability of the game. However, it is also important in applications and other types of software to keep things as efficient as possible.

Anything that makes user wait will detract from productivity of package in general.

Algorithms

The most important key to optimising different routines is the overall approach taken to implementing them in first place. There will always be half a dozen ways of approaching a problem giving half a dozen possible solutions. In programming, it is usually best to pick the solution that will produce the result in the quickest time.

Loops

When looking for ways to optimise a routine the best place to start is to examine the loops (For...Next, While...Wend etc.). Time it takes to perform the code inside a loop is multiplied by number of times it loops. This may seem logical but often programmers will equate the number of lines code in a routine to time taken to execute it.

```
For i=1 to 100
  NPrint "hello"
Next
```

Will take exactly the same amount of time as typing the following 100 times which will equate to 300 lines of code!

```
For i=1 to 1
  NPrint "hello"
Next
```

Once one can visualise loops expanded out, the notion that if anything can be removed from inside a loop to before or after the loop then DO IT!

Look-Up Tables

Replacing numeric functions with look-up tables is an effective way of gaining excellent speed increases. A look-up table or LUT for short, is an array that contains all the possible solutions that the numeric function would be expected to provide.

The most common example of using LUTs for healthy speed increases is when using trig functions such as Sine or Cosine. Instead of calling the Sin function, an array containing a sine wave is created, the size of the array depends on the accuracy of the angle parameter in your program.

If a was an integer variable containing an angle between 0 and 360 we could replace any Sin functions such as x=Sin(a*130/pi) with x=sinlut(a) which will of course be more than 10 times as quick. Such an array would be setup in a program as follows:

```
Dim sinlut(360)
For i=0 To 360
  sinlut(i)=Sin(i*180/pi)
Next
```

Using Pointers

When doing many operations on a particular subfield in a NewType a temporary pointer variable of the same subfield type can be created and that used instead of the larger (and slower) path name:

```
UsePath a(i)\alien\pos
```

replaced by:

```
UsePath *a
*a.pos=a(i)\alien
```

Testing Performance

Often it's important test 2 different routines to see which offers the faster solution. The easiest way is to call each of them 5000 times or so and time which is quicker.

When writing arcades that will be performing a main loop each frame, is useful to poke background colour register before and after a specific routine to see how much of the frame it is using.

The following will show how much of a frame it takes to clear a bitmap:

```
While Joyb(0)=0
  VWait
  Cls
  MOVE #$f00,$dff180            ;poke background colour red
Wend
```

Different colours can be used for different parts of the main loop. Remember that at the top of each slice the background colour will be reset.

Optimising Games

A quality arcade game should always run to a 50th, meaning the main loop always takes less than a frame to execute and so animation etc. are changed every frame giving the game that smooth professional feel.

This time frame means the programmer will often have to sacrifice certain elements in their game and maybe reduce colours and size of shapes to get the main loop fast enough.

The following are several methods for optimising code main loops in games:

- Disable runtime errors in the compiler options when testing speed of code as the error checker slows code dramatically.
- Poke the background colour register with different values between main routines to work out which ones are taking too long:

```
MainLoop:
  VWait
  Gosub movealiens
  MOVE.w #$f00,$dff180            ;turn background red
  Gosub drawaliens
  MOVE.w #$f00,$dff180            ;turn background green
```

- Use QBlits as they are the fastest way of implementing animated graphics in Blitz2.
- If aliens change direction using complex routines, split aliens into groups and every frame select a different group to have their directions changed, the others can move in same direction until it's their turn. This method applies to any routines that don't have to happen every frame but can be spread across several frames in tidy chunks.
- Decrease the size of the display. During a frame, the display slows down processor and blitter. A smaller display increases amount of time given to processor and blitter.

Those with fast mem and faster processors should remember that most people don't have either when testing speed of your code.

Number Guessing

Following is a small program where computer guesses a random number and you have to guess it in less than ten turns.

```
NPrint "I just picked a number from 0 to 100"
NPrint "I'll give you ten turns to guess it:"
a=Rnd(100)
n=1

Repeat
  Print "Attempt #",n," ?"
  b=Edit(10)
  If b=a Then NPrint "Lucky Guess":Goto finish
  If b<a Then NPrint "Too Small"
  If b>a Then NPrint "Too Large"
  n+1
Until n=11

NPrint "Out of turns!"

finish:
NPrint "Press mouse button to exit."

MouseWait
```

First , you'll find it pretty hard to guess the number, this is because the number Blitz generates is not by default an integer and will hence include some fractional part.

Change the line a=Rnd(100) to either a.w=Rnd(100) or a=Int(Rnd(100)).

The .w suffix means the variable a is now a word type (an integer with range –32768...32767). If you use the Int function in the second option, a is still a quick type but the random number has its fractional part chopped. When you use variables in Blitz2 without a.type suffix they default to the quick type which is a number with range -32768...32767 with 1/65536 accuracy. See the Variable Types section for a more in-depth discussion of this topic.

If you want all the variables in the program to default to the integer word type (not quick) then add the following line to the top of the program:

```
DEFTYPE .w ;all variables without suffix default to words
```

As with other BASICs once the variable is used once, its type is defined and future references do not require the .type suffix.

Unlike other BASICs the Print command does not move the cursor to a new line when finished so the command NPrint is used for this.

The Edit() function is used instead of the older input command.

Also the semicolon is used instead of the REM command in Blitz2 and does not retain any of its older functionality in Print statements.

Creating Stand-Alone Workbench Programs

The number guessing program can be made to run from Workbench with its own icon. Add the following lines to the start of your code.

The text after the semicolons are known as remarks. As mentioned, the semicolon in Blitz2 replaces the old REMark command in older BASICs.

```
; Number Guessing Program
WBStartup      ;necessary for program to be run from Workbench

FindScreen 0   ;get front most Intuition screen
Window 0,0,0,320,210,$1000,"Hello World",1,2
```

When you compile & execute the program now, the window replaces the default CLI for input and output.

One thing that you should replace is the b=Edit(10) function to: b=Val(Edit$(10)) This gets rid of default 0 character that appears in window form of Edit() function.

Ensure the Create Executable Icon option in the Compiler Options is set to ON.

Now, select Create Executable from compiler Menu or use the Amiga E shortcut.

Type the name of program you wish to create. You have now created your first stand alone program with Blitz2, go to Workbench and click on new program's icon to test it.

A Graphic Example

The following program opens its own screen and draws what is known as a rosette, a pattern where lines are connected between all the points around a circle.

```
;rosette example
n=20

NEWTYPE .pt
  x.w:y
End NEWTYPE

Dim p.pt(n)

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

Screen 0,25 , hires 1 colour interlace screen
ScreensBitMap 0,0

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

MouseWait
```

The NewType .pt defined in the program has two items or fields x & y. This means that instead of dimming an array of x.w(n) and y.w(n) we can dim one array of p.pt(n) which can hold the same information.

The backslash "\" character is used to access the separate fields of the NEWTYPE . The first For...Next loop assigns the points of a circle into the array of points.

The ScreensBitMap command allows us to draw directly onto screen with Plot, Line, Box and Circle commands. Programs that use windows should not use this method, rather they should draw into specific windows using WPlot, WLine WBox & Wcircle.

Using Menus and the Blitz2 File Requester

The following program opens its own screen & window, attaches a menu list, and depending what user selects from menus, either opens a Blitz file requester or exits.

```
; A Simple File Requester example
Screen 0,11,"Select A Menu"    ;open our own intuition screen
MenuTitle 0,0,"Project"        ;setup a menu list
Menultem 0,0,0,0,"Load ","L"
Menultem 0,0,0,1,"Save ","S"
Menultem 0,0,0,2,"Quit ","Q"
MaxLen path$=192               ;must be called before a file requester is used
MaxLen name$=192

;Set up a BACKDROP (ie - invisible) window
Window 0,0,0,320,200,$1900,"",1,2

WLocate 0,20                   ;move cursor to top left of window
SetMenu 0                      ;attach our menu list to our window

Repeat
  Select WaitEvent
    Case 256 ;it's a menu event!
      Select ItemHit
        Case 0            ;load, its item 0 which means load
```

```
          p$=FileRequest$("FileToLoad",path$, name$)
          NPrint "Attempted to Load ",p$
        Case 1            ;save, it’s item 1 which means save
          p$=FileRequest$("FileToSave",path$, name$)
          NPrint "Attempted to Save ",p$
        Case 2            ;it’s item 2 which means quit
          End
      End Select
  End Select
Forever
```

MaxLen command is used to allocate a certain amount of memory for a string variable in Blitz2. This is necessary so that the two string variables required by the file requester command are large enough for the job.

Menus created by the MenuTitle and MenuItem commands are attached to the Window using the SetMenu command.

The Select...Case...End Select structures are the best way of handling information coming from a user. When the user selects a menu, closes a window or clicks on a gadget an ‘event' is sent to the program. Usually an application program will use WaitEvent which makes program sleep until the user does something. When multitasking, a program that is asleep will not slow down the execution of other programs running.

Once an event is received, the event code returned by WaitEvent specifies what type of an event occurred. For example, a menu event returns 256 ($100 hex) and a close window event returns 512 ($200 hex).

String Gadgets

The following program demonstrates the use of string gadgets. These allow the user to enter text via the keyboard via three string gadgets for decimal, hex and binary I/O.

When the user types a number into one of the gadgets and hits return, the program receives a gadgetup event. The GadgetHit function returns which gadget caused the event. The program then converts the number the user typed into other number systems (decimal, hex or binary) and displays the results in each of the string gadgets.

The ActivateString command means the user does not need to click on the gadget to reactivate it so that they can type in another number.

```
;decimal hex binary converter
FindScreen 0
StringGadget 0,64,12,0,0,18,144
StringGadget 0,64,26,0,1,18,144
StringGadget 0,64,40,0,2,18,144

Window 0,100,50,220,56,$1008,"BASE CONVERTER",1,2,0
WLocate 2,04:Print "DECIMAL"
WLocate 2,18:Print " HEX$"
WLocate 2,32:Print "BINARY%"
DEFTYPE.l value

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

    ActivateString 0,GadgetHit
    SetString 0,0,Str$(value)
    SetString 0,1,Right$(Hex$(value),4)
    SetString 0,2,Right$(Bin$(value),16)
    Redraw 0,0:Redraw 0,1:Redraw 0,2

  Endif
Until ev=$200
```

Prop Gadgets

The following program creates a simple RGB palette requester allowing user to adjust the colours of the screen. PropGadgets can be thought of as sliders. In this example we create three vertical PropGadgets to represent the Red, Green and Blue components of the current colour register selected.

The 32 colour registers are represented with 32 text gadgets. The gadget's colour is set by changing GadgetPens before the gadget is added to the gadget list. Using GadgetJam 1 the two spaces are shown as a block of colour.

```
; simple palette requester
FindScreen 0
```

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

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

Window 0,100,50,300,72,$100A,"PALETTE REQUESTER",1,2,0
cc=0:Toggle 0,3+cc,0n: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,0n:Redraw 0,3+cc
    cc=GadgetHit-3
    Toggle 0,3+cc,0n: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
```

Database Type Application

The following listing is a simple data base program to hold a list of names, phone numbers and addresses.

The user interface can either be typed in as listed or created using the Intuition Tools tutorial later in this manual.

If a text file exists called phonebook.data we read it into a list. Each item of the list has been set up to hold 4 strings using the NewType person. Using a list instead of a normal array means that we think of each record inside the list as connected to the one before and the one after rather than just being an individual item. Blitz2 keeps an internal pointer to the current item and the various list commands enable us to change that internal pointer and operate on the item it points to.

Phone Book Program

```
FindScreen 0
;the following is from ram:t as created in the intuition tools tutorial
Borders On:BorderPens 1,2:Borders 4,2
StringGadget 0,72,12,0,1,40,239
StringGadget 0,72,27,0,2,40,239
StringGadget 0,72,43,0,3,40,239
StringGadget 0,72,59,0,4,40,239
GadgetJam 0:GadgetPens 1,0
TextGadget 0,8,75,0,10,"NEW ENTRY"
TextGadget 0,97,75,0,11,"-1 <"
TextGadget 0,129,75,0,12, "<<"
TextGadget 0,161,75,0,13, ">>"
TextGadget 0,193,75,0,14,">1 "
TextGadget 0,226,75,0,15,"DIAL"
TextGadget 0,270,75,0,16,"LABEL"
SizeLimits 32,32,-1,-1

Window 0,0,24,331,91,$100E,"MY PHONE BOOK",1,2,0
WLocate 2,19:WJam 0:WColour 1,0:Print "Address"
WLocate 19,50:Print "Phone"
WLocate 27,3:Print "Name"
; and now we start typing...
#num=4 ;4 strings for each person

NEWTYPE .person
  info$[#num]
End NEWTYPE

Dim List people.person(200)
USEPATH people()
; read in names etc from sequential file

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

ResetList people()
;if empty add blank record
If NOT NextItem(people()) Then AddItem people()
refresh:
ref=0

For i=0 To #num-1
  SetString 0,i+1,\info[i]:Redraw 0,i+1
```

```
Next

ActivateString 0,1 :VWait 5
Repeat
  ev.l=WaitEvent

  If ev=$200                ;close window event
    Gosub update
    If WriteFile(0,"phonebook.data");save data to file
      FileOutput 0
      ResetList people()
      While NextItem(people())
        For i=0 To #num-1 :NPrint \info[i]:Next
      Wend
      CloseFile 0
    Endif
  Endif

  If ev=64
    If GadgetHit=#num Then ActivateString 0,1
    If GadgetHit<#num Then ActivateString 0,GadgetHit+1
    Select GadgetHit
      Case 10
        Gosub update:If AddItem(people()) Then ref=1
      Case 11
        Gosub update:If FirstItem(people()) Then ref=1
      Case 12
        Gosub update:If PrevItem(people()) Then ref=1
      Case 13
        Gosub update:If NextItem(people()) Then ref=1
      Case 14
        Gosub update:If LastItem(people()) Then ref=1
    End Select
  Endif
Until ref=1

Goto refresh
Update:
For i=0 To #num-1 : \info[i]=StringText$(0,i+1) : Next : Return
```

List Processor for Exec Based Lists

Following is an example of accessing OS structures. Before entering this program you will need to add the AmigaLibs.res file to the Blitz 2 environment. To do this open the Compiler Options requester from the Compiler Menu. Click in the Residents box and type in amigalibs.res.

By selecting ViewTypes from the compiler menu the entire set of structs should be listed that are used by the Amiga's operating system.

The first line of our program defines the variable exec as a pointer to type ExecBase. As the Amiga keeps the location of this variable in memory location 4 we can use the Peek.l (long) command to read the 4 byte value from memory into our pointer variable.

Blitz2 now knows that exec points to an execbase structure and using the backslash character we can access any of the variables in this structure by name.

If you select ViewTypes from the compiler menu and type in ExecBase (case sensitive) you can view all the variables in the execbase structure.

We then define another pointer type called *mylist.List. We can then use this to point to any List found in execbase such as LibList or DeviceList.

An exec list consists of a header node and a series of link nodes that hold the list of devices or libraries etc.

We point mynode at the lists first link node in the third line of code.

The next line loops through the link nodes until the node's successor=0 which means we have arrived back at the header node.

Peek$ reads ASCII data from memory until a zero is found, this is useful for placing text pointed to by a C definition such as *In_Name.b into Blitz 2's string work area.

We then point mynode at the next node in the list.

Exec List Processor

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

While *mynode\ln_Succ
  a$=Peek$(*mynode\1n_Name)
  NPrint a$
  Amynode=*mynode\ln Succ
Wend

MouseWait
```

Prime Number Generator

Following program generates a list of prime numbers from 2 to a limit specified by user. A list of all the prime numbers found is kept in a Blitz 2 List structure.

We begin by inputting the upper limit from the user using the default input output and the edit() command, the numeric form of the edit$() command.

A While...Wend structure is used to loop through the main algorithm until upper limit is reached. The algorithm simply takes next integer, loops through the list of prime numbers it has already generated until either finds a divisible number or it is too far through the list (item in list is greater than square root of number being checked).

If the algorithm does not find a divisor in its search through the list it prints the new prime and adds it to the end of the list.

```
Print "Primes to what value "      ;find out limit to run program to
v=Edit(80)                         ;input numeric
If v=0 Then End                    ;if 0 then don't carry on
tab.w=0 : tot.w=0                  ;reset counters
Dim List primes(v)                 ;dim a list to hold primes
p=2                                ;add the number2 to our list
AddItem primes()
primes()=p

While p<v                          ;loop until limit reached
  p+1                              ;increment p
  flag=1                           ;set flag
  d=0
  q=Sqr(p)                         ; set search limit
  ResetList primes()               ;loop through list

  While NextItem(primes()) AND d<q AND flag
    d=primes()
    flag=p MOD d
  Wend

  If flag<>0                       ;if found print and add it to list
    Print p,Chr$(9)                ;chr$(9) is a TAB character
    tab+1:tot+1
    If tab=10 Then NPrint "":tab=0
    AddLast primes()
    primes()=p
  Endif
Wend

NPrint Chr$(10)+"Found ",tot," Primes between 2 & ",v
NPrint "Left Mouse Button to Exit"

MouseWait
```

Clipped Blits

The Following program illustrates a method to clip blits. When a shape is blitted outside the area of a bitmap an error occurs. To have shapes appear half inside a bitmap and half outside we use a larger bitmap and position the display inside. The size of the outer frame is dependent on the size of the shapes that will be drawn.

In the following example we are using a 32x32 pixel shape so we need an extra 32 pixels all round the bitmap. The Show 0,32,32 command centres the display inside the larger bitmap. We also have to use the extended form of the slice command as we are displaying a bitmap wider than the display.

The RectsHit(x,y,1,1,0,0,320+32,256+32) function returns true if the shape is inside the larger bitmap and should be blitted. If the shape was larger or it had a centred handle the parameters would need to be changed to accommodate these factors.

The makeshape routine creates a temporary bitmap to draw a pattern and then transfer it to a shape object using the GetaShape command.

```
Blitz
Gosub makeshape
BitMap 0,320+64,256+64,3
Slice 0,44,320,256,$fff8,3,8,8,320+64,320+64
Show 0,32,32

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

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

Return
```

Dual Playfield Slice

The following program demonstrate use of a dual playfield display. As described in a previous chapter dual playfield lets us display two bitmaps simultaneously using ShowF and ShowB commands.

The macro rndpt simply inserts the code Rnd((i40),Rnd(512) into source each time it is called. For instance Line !rndpt,!rndpt,Rnd(7) is expanded internally by compiler to:

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

Once again the extended from of the slice command has to be used with flags set to $fffa giving us a lores dual playfield scrollable display.

In dual playfield we can think of having two displays. The ShowF command positions the front display inside BitMap 1, the ShowB command positions the backdrop display inside BitMap 0. Note that we must pass the x position of the other display with ShowF and ShowB so that Blitz2 can calculate internal variables properly.

```
Blitz
Macro rndpt Rnd(640),Rnd(512):End Macro
BitMap 0,640,512,3

For i=0 To 255
  Line !rndpt,!rndpt,Rnd(7)
Next

BitMap 1,640,512,3

For i=0 To 255
  Circlef!rndpt,Rnd(15),Rnd(7)
Next

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

While Joyb(0)=0
  VWait
  x1=160+Sin(r)*160
  y1=128+Cos(r)*128
  x2=160-Sin(r)*160
  y2=128-Cos(r)4128
  ShowF 1,x1,y1,x2
  ShowB 0,x2,y2,x1
  r+.05
Wend
```

Double Buffering

The following code illustrates the use of a double buffered display which is necessary to achieve smooth moving graphics. The trick with double buffering is that while one bitmap is displayed we can change the other without any glitches happening on the display.

The VWait command waits for the vertical beam to be at the top of the display which is when we are allowed to swap the bitmaps being displayed without getting any glitches.

The db=1-db equation will mean that db alternates between 0 & 1 each frame. We Show db, toggle it (db=1-db) and then Use Bitmap db to achieve the "draw to one bitmap while displaying the other" technique known as double buffering.

Because we have two bitmaps, we need two queues to use QBlit properly. QBlit works by doing a normal Blit and storing the position of Blit in a queue. The UnQueue command will erase all parts of the screen listed in the queue so we can draw the balls in their new positions without leaving "trails" behind them from their old position.

The move #-1,$dffl80 line pokes the background colour to white, this allows us to see how much of the frame has been taken since VWait to execute the code. If we increase the number of balls, the moving and drawing loop will take longer than a frame (50th of a second) and the white will start flashing as the poke will only be happening every second frame. See chapter 10 for more thorough discussion of frame rates etc.

The only other thing I'll mention is the bounce logic used when the ball moves outside the bitmap. We reverse the direction but also add the new direction to the position so the program never attempts to Blit the shape outside of the bitmap.

```
Blitz
n=25

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

Dim List b.ball(n-1)

While AddItem(b())
  b()\x=Rnd(320-32),Rnd(256-32),Rnd(4)-2,Rnd(4)-2
Wend

Gosub getshape

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

While Joyb(0)=0
  VWait
  Show db
  db=1-db
  Use BitMap db
  UnQueue db
  ResetList b()
  USEPATH b()
  While NextItem(b())
    \x+\xa:\y+\ya
    If NOT RectsHit(\x,\y,1,1,0,0,320-32,256-32)
      \xa=-\xa:\ya=-\ya
      \x+\xa:\y+\ya
    Endif
    QBlit db,0,\x,\y
  Wend
  MOVE #-1,$dff180
Wend
End
```

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

Smooth Scrolling

This final example demonstrates smooth scrolling as discussed in a previous chapter.

Scroll commands are used to copy the left side of the bitmap to right and the top half of the bitmap to bottom. This effect means large bitmap is the same in each quarter.

Because of this we can scroll display across the bitmap, and when hit the right edge reset the display back to the left edge without any jump in display as both left and right sides of the bitmap are the same. This is same for scrolling display down the bitmap.

To be able to access mouse moves we need the Mouse On command. We can then take the amount the mouse has been moved by the user and add it to the speed in which we are moving the display around the bitmap.

The QLimit(xa+MouseXSpeed,-20,20) command makes sure that the xa (x_add) variable always stays inside the limits -20...20.

The x=QWrap(x+xa,0,320) command means that when the displays position inside the bitmap reached the right edge of the bitmap it wraps around to the left.

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

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

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

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

11. THE DISPLAY LIBRARY & AGA

Introduction

Display Library is a recent addition to Blitz. Developed as a replacement to Slices it not only offers games programmers access to all of new AGA features but offers a slightly more modular approach to controlling the Amiga's graphics hardware.

The Amiga's display is controlled by the copper. The copper is a secondary processor that executes a list of instructions every frame. For those new to such concepts, the Amiga redraws the screen 50 times a second and each redraw is known as a frame. The video beam which sweeps across the screen drawing each pixel is controlled by certain hardware registers, these registers are poked by the copper whose job it is to keep everything in sync. A coplist contains information about colours, bitplanes, sprites, resolution and more that the video beam requires to render a typical display.

Initialising

Unlike Slices which appear as soon as they are initialised the display library requires coplists to be initialised (using InitCopList) prior to a display being created (using CreateDisplay). The important difference here is that Slices require memory to be allocated each time a change to the video display is required while the Display library allows multiple CopLists to be initialised before any displays are created.

There are two forms of the InitCopList command. The short version simply requires the CopList# which is to be initialised and the flags. The height of the display will default to 256 pixels high. A width of 320, 640 or 1280 will be used depending on the resolution set in the flags as will the number of colours.

The longer version has the following format:

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

The ypos value is usually set to 44, the standard top of frame for a PAL display. If CopLIst is to be used below another coplist on the same display ypos should be set to 2 scan lines below the last CopLists bottom line.

Sprites should always be set to eight, even if they are not all available, colours should be set to the number required. When using more than 32 colours ensure that the #agacolors flag MUST be set.

Customs allocate enough room for advanced custom copper lists to be attached to each display. See later on in this chapter for a discussion on using custom cops.

Flags Used With InitCopList

The flags value is calculated by adding the following values together.

Note: Variables must be long (32 bits) when used as the flags parameter for the InitCoplist command.

#onebitplane	= $01	
#twobitplanes	= $02	
#threebitDlanes	= $03	
#fourbitplanes	= $04	
#fivebitplanes	= $05	
#sixbitplanes	= $06	
#sevenbitplanes	= $07*	
#eightbitplanes	= $08*	
#smoothscrolling	= $10	;set if you will be scrolling the bitmap
#dualplavfield	= $20	;enable dual playfield mode
#extrahalfbrite	= $40	;forces 6 bitplane display into ehb mode
#ham	= $80	;display in ham
#lores	= $000	
#hires	= $100	
#superhires	= $200	
#loressprites	= $400	
#hiressprites	= $800*	
#superhiressprites	= $c00	
#fetchmode0	= $0000	
#fetchmode1	= $1000*	
#fetchmode2	= $2000*	
#fetchmode3	= $3000*	
#agacolors	= $10000*	

* These flags should only be used with AGA Amigas.

Colours

The #agacolors flag must ALWAYS be set when more than 32 colours are in use or when 24 bit colour definition is required.

SmoothScrolling

By setting the smooth scrolling flag the extended form of DisplayBitmap may be used which allows the bitmap to be displayed at any offset. This enables the programmer to scroll the portion of the bitmap being displayed. See the Blitz Mode programming chapter for an explanation of hardware scrolling.

Notes:

- Always use the extended form of DisplayBitmap with smoothscrolling set, even when the offset is 0,0.
- DisplayBitmap accepts quick types for the x offset and will position the bitmap in fractions of pixels on AGA machines.
- The width of the display will be less than the default 320/640/1280 when smooth scrolling is enabled.

Dual Playfield

By setting the DualPlayfield flag two bitmaps may be displayed on top of each other in one display. A combination of DualPlayfield and SmoothScrolling is allowed for parallax type effects. Note that with AGA machines, it is possible to display two 16 colour bitmaps by enabling DualPlayfield and setting number of bitmaps to 8.

Sprites

The number of sprites available will depend on the type of display and the fetch mode settings. Most AGA modes will require the display to be shrunk horizontally for 8 sprites to be displayed. Currently this can only be achieved using the DisplayAdjust command and certain examples of this can be found on the Blitz examples disk.

AGA hardware allows the programmer to display sprites in lores, hires or superhires. The higher resolutions allow graphics dithering by the artist, essential if 3 coloured sprites are in use. Larger sprites are also available using the SpriteMode command. Dithered large, super hi-res sprites can be created to look better than lower resolution 16 colour sprites using such tools as ADPro.

Note that it is unrealistic to display more than 4 bitplanes and have more than 3 sprite channels available, the adjust required results in a very narrow display indeed.

Fetch Mode

AGA hardware allows bitplane data to be fetched by the DMA in 16,32 or 64 pixel groups. The larger fetches give the processor more bandwidth, this is especially noticeable with AGA Amigas running without additional fast mem.

Using increased fetchmodes, bitplanes must always be a multiple of 64 pixels wide.

Those wanting to attempt DisplayAdjusts on displays with larger fetch modes will encounter severe difficulties in creating a proper display. We think it is actually impossible for displays to run at fetch mode 3 with more than 1 sprite without having to adjust the display to around 256 pixels across.

Multiple Displays

When more than one CopList is to be displayed, care must be taken that there is a gap of at least 3 lines between each. This means the ypos of a lower coplist must be equal or greater than the above ypos+height+3.

Advanced Copper Control

The long format of the InitCopList command allows allocation for custom copper commands. Certain commands have been added to the Display Library which will require this parameter to be set.

There are two forms of custom copper commands. The first will allow the copper to affect the display every scan line while the second defines a certain line for the copper to do its thing. These new commands include:

The following require a negative size. This denotes that so many instructions must be allocated for every scan line of the display.

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

The following require the size be specified as a positive parameter denoting that so many instructions be allocated for each instance of each command. Note that these two commands may NOT be mixed with the commands above.

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

Use of these commands is illustrated by code included in Blitz examples drawer.

Display Example 1

This first example creates two large bitmaps. It renders lines to one and boxes on the other. A 32 colour palette is created and the first 16 colours are used by the back playfield and second 16 by the front playfield.

The flags in the InitCopList command are the sum of the following:

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

InitCopList can be executed before going into Blitz mode. All display commands are mode independent except CreateDisplay which can only be executed in Blitz mode.

Finally, note the extended form of the DisplayBitmap command. This allows the offset position of both bitmaps to be assigned with the one command.

```
; two 16 colour playfield in dualplayfield mode
BitMap 0,640,512,4
BitMap 1,640,512,4

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

InitPalette 0,32

For i=1 To 51 : AGAPalRGB 0,i,Rnd(256),Rnd(256),Rnd(256):Next

InitCopList 0,$13038
Blitz
CreateDisplay 0
DisplayPalette 0,0

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

End
```

Display Example 2

This second example demonstrates the use of sprites on a Display. DisplayAdjust is required to allow us access to all 8 sprite channels. Unfortunately it is difficult to up the fetch mode in this example without resorting to a very thin display.

SpriteMode 2 tells Blitz to create 64 pixel wide sprites for each channel. Each sprite would require 4 channels without SpriteMode, one of the better new features of AGA.

It should be noted also that the DisplaySprite command also accepts fractional x parameters and will position the sprite at fractional pixel positions if possible.

```
; smoothscrolling 16 colour screen with 8x64 wide sprites
SpriteMode 2
InitShape 0,64,64,2:ShapesBitMap 0,0
Circlef 32,32,32,1:Circle7 16,8,6,2:Circlef 48,8,6,3:Circlef 32,32,8,0
GetaSprite 0,0
```

```
BitMap 0,640,512,4

For i=0 To 100
  Use BitMap 0:Box Rnd(640),Rnd(512),Rnd(640),Rnd(512),Rnd(16)
Next
InitPalette 0,48

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

InitCopList 0,$10014
DisplayAdjust 0,-2,8,0,16,0 ;under scan!
Blitz
CreateDisplay 0
DisplayPalette 0,0

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

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

End
```

R-1: PROGRAM FLOW

A computer program is made up of a sequence of commands that are executed sequentially (one after the other). Certain commands are used to interrupt this process and cause program execution to jump to a different location in the program. There are several different ways of controlling this program flow in Blitz.

BASIC commands to change program flow such as Goto and Gosub are standard in Blitz. Unlike older BASICs, locations are specified as program labels not line numbers. Modern BASIC features such as procedures (Statements & Functions), While...Wend, Repeat...Until, Select...Case allow a more structured approach to programming.

Finally Blitz allows control over interrupts. This allows external events to override normal program flow and jump (temporarily) to a predefined location in the program.

Goto label

Causes program flow to be transferred to the specified program label. This allows sections of a program to be skipped or repeated.

Gosub label

Operates in two steps. First, the location of the instruction following the Gosub is remembered in a special storage area (known as the 'stack'). Secondly, program flow is transferred to the specified program label.

The section of program that program flow is transferred to is known as a 'subroutine' and should be terminated by a Return command.

Return

Used to return program flow to the instruction following the previously executed Gosub command. This allows the creation of 'subroutines' which may be called from various points in a program.

On expression Goto|Gosub label[,label...]

Allows 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 specified expression.

If the specified expression results in a 1, then the first label will be branched to. A result of 2 will cause the second label to be branched to and so on. If the result of the expression is less than one, or not enough labels are supplied, then the program flow will continue without a branch.

End

Halt program flow completely. In case of programs run from Blitz editor, you will be returned to editor. In case of executable files, will be returned to Workbench or CLI.

Stop

Causes the Blitz debugger to interrupt program flow. Place Stop commands in your code as breakpoints when debugging and ensure runtime errors are enabled. Click on Run from the debugger to continue program flow after a Stop.

If expression [Then...]

Allows execution of a section of program depending on the result of an expression. The Then command indicates only the rest of the line will be defined as the section of code to either execute or not. Without a Then the section of code will be defined as that up to the EndIf command.

Endif

Used to terminate an 'If' block. An If block is begun by use of If statement without the Then present. Please refer to If for more information on If blocks.

Else [Statement...]

May be used after an If to cause program instructions to be executed if the expression specified in the If proved to be false.

While expression

Used to execute a series of commands repeatedly while the specified expression proves to be true. The commands to be executed include all the commands following the While until the next matching Wend.

Wend

Used in conjunction with While to determine a section of program to be executed repeatedly based upon the truth of an expression.

Select expression

Examines and 'remembers' the result of the specified expression. The Case commands may then be used to execute different sections of program code depending on the result of the expression in the Select line.

Case expression

Used following a Select to execute a section of program code when, and only when, the expression specified in the Case statement is equivalent to the expression evaluated in the Select statement.

If a Case statement is satisfied, program flow will continue until the next Case Default or End Select statement is encountered, at which point program flow will branch to the next matching End Select.

Default

May appear following a series of Case statements to cause a section of code to be executed if NONE of the Case statements were satisfied.

End Select

Terminates a Select...Case...Case...Case sequence. If program flow had been diverted through the use of a Case or Default statement, it will continue from the terminating End Select.

For var=expression1 To expression2 [Step expression3]

The For statement initialises a For...Next loop. All For...Next loops must begin with a For statement, and must have a terminating Next statement further down the program. For...Next loops cause a particular section of code to be repeated a certain number of times. The For statement does most of the work in a For...Next loop. When For is executed, the variable specified by var (known as the index variable) will be set to the value expression1. After this, the actual loop commences.

At the beginning of the loop, a check is made to see if the value of var has exceeded expression2. If so, program flow will branch to the command following For...Next loop's Next, ending the loop. If not, program flow continues on until the loop's Next is reached. At this point, value specified in expression3 (‘step’ value) is added to var, and program flow is sent back to the top of the loop, where var is again checked against expression2. If expression3 is omitted, a default step value of 1 will be used.

In order for a For...Next loop to count ‘down’ from one value to a lower value, a negative step number must be supplied.

Next [var[,var...]]

Terminates a For...Next loop. Please refer to the For command for more information on For...Next loops.

Repeat

Used to begin a Repeat...Until loop. Each Repeat statement in a program must have a corresponding Until further down the program.

The purpose of Repeat/Until loops is to cause a section of code to be executed AT LEAST ONCE before a test is made to see if the code should be executed again.

Until expression

Used to terminate a Repeat...Until loop. If Expression is true (non 0) program flow will continue from the command following Until. If expression is false (0) program flow will go back to the corresponding Repeat, found further up the program.

Forever

May be used instead of Until to cause a Repeat...Until loop to NEVER exit. Executing Forever is identical to executing 'Until 0'.

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

Sometimes, it may be necessary to exit from a particular type of program loop in order to transfer program flow to a different part of program. Pop must be included before the Goto which transfers program flow out from the inside of the loop.

Actually, Pop is only necessary to prematurely terminate Gosubs, Fors and Selects. If, While and Repeat have been included for completeness but are not necessary.

MouseWait

Halts the program until left mouse button is pushed. If left mouse is already held down when a MouseWait is executed, program will simply continue through.

MouseWait should normally be used only for program testing purposes, as MouseWait severely slows down multi-tasking.

VWait [frames]

Causes program flow to halt until the next vertical blank occurs. Optional frames parameter may be used to wait for a particular number of vertical blanks.

Especially useful in animation for synchronising display changes with the rate at which the display is physically redrawn by the monitor.

Statement procedurename{[parameter1[,paramater2...]]}

Declares all following code up to the next End Statement as being a 'statement type' procedure.

Up to 6 parameters may be passed to a statement in the form of local variables through which calling parameters are passed.

In Blitz, all statements and functions must be declared before they are called.

End Statement

Declares the end of a ‘statement type’ procedure definition. All statement type procedures must be terminated with an End Statement.

Statement Return

May be used to prematurely exit from a ‘statement type’ procedure. Program flow will return to the command following the procedure call.

Function [.type] procedurename{[parameter1[,parameter2...]]}

Declares all following code up to the next End Function as being a function type procedure. The optional type parameter may be used to determine what type of result is returned by the function. Type, if specified, must be one Blitz’s 6 primitive variable types. If no type is given, the current default type is used. Up to 6 parameters may be passed to a function in form of local variables through which calling parameters are passed. Functions may return values through the Function Return command. In Blitz, all statements and functions must be declared before they are called.

End Function

Declares the end of a ‘function type’ procedure definition. All function type procedures must be terminated with an End Function.

Function Return expression

Allows ‘function type' procedures to return values to their calling expressions. Function type procedures are called from within Blitz expressions.

Shared var[,var...]

Used to declare certain variables within a procedure definition as being global variables. Any variables appearing within a procedure definition that do not appear in a Shared statement are, by default, local variables.

SetInt type

Used to declare a section of program code as 'interrupt' code. Often, when a computer program is running, an event of some importance takes place which must be processed immediately. Different types of interrupt on the Amiga are as follows:

Type	Cause of Interrupt
0	Serial transmit buffer empty
1	Disk block read/written
2	Software interrupt
3	Cia ports interrupt
4	Co-processor ('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

The most useful of these interrupts is the vertical blank interrupt. This interrupt occurs every time an entire video frame has been fully displayed (about every fiftieth of a second), and is very useful for animation purposes. If a section of program code has been designated as a vertical blank interrupt handler, then that section of code will be executed every fiftieth of a second.

Interrupt handlers must perform their task as quickly as possible, especially in case of vertical blank handlers which must NEVER take longer than 1/5 of sec. to execute.

Interrupt handlers in Blitz must NEVER access string variables or literal strings. In Blitz mode, this is the only restriction on interrupt handlers. In Amiga mode, no blitter, Intuition or file I/O commands may be executed by interrupt handlers.

To set up a section of code to be used as an interrupt handler, you use the SetInt command followed by the actual interrupt handler code. An End SetInt should follow the interrupt code. The type parameter specifies the type of interrupt, from the above table, the interrupt handler should be attached to. For example, SetInt 5 should be used for vertical blank interrupt code.

More than one interrupt handler may be attached to a particular type of interrupt.

End SetInt

Must appear after a SetInt to signify the end of a section of interrupt handler code. Please refer to SetInt for more information of interrupt handlers.

ClrInt type

May be used to remove any interrupt handlers currently attached to specified interrupt type. SetInt is used to attach interrupt handlers to particular interrupts.

SetErr

Allows you to set up custom error handlers. Program code which appears after the SetErr command will be executed when any Blitz runtime errors are caused. Custom error code should be ended by an End SetErr.

End SetErr

Must appear following custom error handlers installed using SetErr. Please refer to SetErr tor more information on custom error handlers.

ClrErr

May be used to remove a custom error handler set up using SetErr.

ErrFail

May be used within custom error handlers to cause a ‘normal’ error.

The error which caused the custom error handler to be executed will be reported and transfer will be passed to direct mode.

R-2: VARIABLE HANDLING

To keep track of numbers and text program variables are required. These variables are assigned a name and given a type which dictates the sort of information they are able to contain. Blitz supports 5 standard numeric types and the string type which is used to store text type information.

Variable ‘arrays’ are used to store a large collection of values all of one type, these arrays are similar to normal variables except they must be dimensioned (the number of elements defined) before they are used.

Blitz offers many extensions to these BASIC features. NewTypes may be defined which are a collection of several standard types. A single NewType variable can contain an assortment of numeric and string information similar to structures in C.

List arrays offer more control over standard arrays, they are also much faster to manipulate. Blitz contains many commands for operating on linked lists of data.

Let var=expression

Let is an optional command used to assign a value to a variable. Let must always be followed by a variable name and an expression. An equals sign (=) is placed between the variable name and the expression. If the equals sign is omitted, then an operator (eg. +, *) must appear between the variable name and the expression. In this case, the specified variable will be altered by the specified operator and expression.

Exchange var,var

Will swap the values contained in the two specified variables. May only be used with variables of the same type.

MaxLen stringvar=expression

Sets aside a block of memory for a string variable to grow into. This is normally only necessary in the case of special Blitz commands which require this space to be present before execution. Currently, only two Blitz commands require the use of MaxLen - FileRequest$ and Fields.

DEFTYPE .typename [var[,var...]]

May be used to declare a list of variables as being of a particular type. In this case, var parameters must be supplied.

May also be used to select a default variable type for future ‘unknown' variables. Unknown variables are variables created with no typename specifier. In this case, no var parameters are supplied.

NEWTYPE .typename

Creates a custom variable type and must be followed by a list of entry names separated by ':' and/or new lines. NEWTYPE terminates using End NEWTYPE .

SizeOf .typename[,entrypath]

Allows you to determine the amount of memory, in bytes, a particular variable type takes up. May also be followed by an optional entrypath in which case the offset from the start of the type to the specified entry is returned.

Dim arrayname [list] (dimension1[,dimension2...])

Used to initialise a BASIC array. Blitz supports two array types: simple arrays and list arrays. The optional list parameter, if present, denotes a list array. Simple arrays are identical to standard BASIC arrays, and may be of any number dimensions. List arrays may be of only one dimension.

ResetList arrayname()

Used in conjunction with a list array to prepare the list array for NextItem processing. After executing a ResetList, the next NextItem executed will set the list array's 'current element' pointer to the list array's very first.

ClearList arrayname()

Used in conjunction with list arrays to completely 'empty' out the specified list array. List arrays are automatically emptied when they are created.

AddFirst(arrayname())

Allows you to insert an array list item at the beginning of an array list. Returns a true/false value reflecting whether or not there was enough room in the array list to add an element. If an array element was available, AddFirst returns a true value (-1), and sets the list array's current item pointer to the item added. If no array element was available, AddFirst returns false (0).

AddLast(arrayname())

Allows you to insert an array list item at the end of an array list. AddLast returns a true/false value reflecting whether or not there was enough room in the array list to add an element. If an array element was available, AddLast returns a true value (-1), and sets the list array's current item pointer to the item added. If no array element was available, AddLast returns false (0).

AddItem(arrayname())

Allows you to insert an array list item after the list array's current item. AddItem returns a true/false value reflecting whether or not there was enough room in the array list to add an element. If an array element was available, AddItem returns a true value (-1), and sets the list array's current item pointer to the item added. If no array element was available, AddItem returns false (0).

KillItem arrayname()

Deletes the specified list array's current item. After executing KillItem, the list array's current item pointer will be set to the item before the item deleted.

PrevItem(arrayname())

Will set the specified list array's current item pointer to the item before the list array's old current item. This allows for backwards processing of a list array. PrevItem returns a true/false value reflecting whether or not there actually was a previous item. If a previous item was available, PrevItem will return true (-1). Otherwise, PrevItem will return false (0).

NextItem(arrayname())

Sets the specified list array's 'current item' pointer to the item after the list array's old current item. This allows for forward processing of a list array. NextItem returns a true/false value reflecting whether or not there actually was a next item available or not. If an item was available, NextItem will return true (-1), otherwise, false (0) is returned.

FirstItem(arrayname())

This will set the specified list array's 'current item' pointer to the very first item in the list array. If there are no items in the list array, FirstItem will return false (0) otherwise, FirstItem will return true (-1).

LastItem(arrayname())

This will set the specified list array's 'current item' pointer to the very last item in the list array. If there are no items in the list array, LastItem will return false (0), otherwise LastItem will return true (-1).

PushItem arrayname()

This causes the specified list array's 'current item' pointer to be pushed onto an internal stack. This pointer may be later recalled by executing PopItem. The internal item pointer stack is set for up to 8 'pushes'.

PopItem arrayname()

This ‘pops’ or ‘recalls’ a previously pushed current item pointer for specified list array. arrayname() must match the array name of most recently executed PushItem.

ItemStackSize maxitems

Determines how many ‘list’ items may be pushed (using PushItem), before items must be Pop’d off again. For example, executing ItemStackSize 1000 will allow you to push up to 1000 list items before you run out of item stack space.

SortList arrayname()

Used to rearrange the order of elements in a Blitz linked list. Order in which the items are sorted depends on the first field of the linked list type which must be a single integer word. Sorting criteria will be extended in future.

Sort arrayname()

Causes the specified array to be sorted. The direction of the sort may be specified using either the SortUp or SortDown commands. The default direction used for sorting is ascending – ie. array elements are sorted into a ‘low to high’ order.

SortUp

Used to force the Sort command to sort arrays into ascending order. Means that after being sorted, an array’s contents will be ordered in a ‘low to high’ manner.

SortDown

Used to force the Sort command to sort arrays into descending order. Means that, after being sorted, an array's contents will be ordered in a ‘high to low’ manner.

R-3: INPUT/OUTPUT

Input/Output is essential for programs to function. Input includes reading data from both disk files and data statements and getting input from the user. Output options include writing data to files, displaying information on the screen and so-on.

Input and output are most commonly achieved with the Edit and Print commands with Edit replacing the standard BASIC Input nomenclature. An assortment of commands are available to redirect input and output to and from files and windows etc. Refer to the File and Window handling sections for more information.

Those developing games in Blitz should refer to the Blitz IO section for input & output commands more suited to their particular requirements.

Print expression[,expression...]

Allows you to output either strings or numeric values to the current output channel. Commands such as WindowOutput or BitMapOutput may be used to alter the current output channel.

NPrint expression[,expression...]

Allows you to output either strings or numeric values to the current output channel. Commands such as WindowOutput or BitMapOutput may be used to alter the current output channel.

After all expressions have been output, NPrint automatically prints a newline character.

Format formatstring

Allows you to control the output of any numeric values by the Print or NPrint commands. FormatString is an 80 character or less string expression used for formatting information by the Print command. Special characters in FormatString are used to perform special formatting functions. These special characters are:

Char	Format Effect
#	If no digit to print, insert spaces into output
0	If no digit to print, insert zeros ('0') into output
.	Insert decimal point into output
+	Insert sign of value
-	Insert sign of value, only if negative
,	Insert commas every 3 digits to left of number

Any other characters in FormatString will appear at appropriate positions in the output. Format also affects the operation of the Str$ function.

FloatMode mode

Allows you to control how floating point numbers are output by the Print or NPrint commands.

Floating point numbers may be displayed in one of two ways: exponential format or standard format. Exponential format displays a FP number as a value multiplied by ten raised to a power. For example 10240 expressed exponentially is '1.024E+4' ie: 1.024 x 10 to the power of 4. Standard format simply prints values 'as is'.

A mode parameter of 1 will cause floating point values to ALWAYS be displayed in exponential format. A mode parameter of -1 will cause FP values to ALWAYS be displayed in standard format. A mode parameter of 0 will cause Blitz to take a 'best guess' at the most appropriate format to use. This is the default mode for FP output.

Note that if Format has been used to alter numeric output, standard mode will always be used to print floating point numbers.

Data[.type] item[,item...]

The Data statement allows you to include pre-defined values in programs and data items may be transferred into variables using the Read statement. When data is read into variables, the type of the data being read MUST match the type of the variable it is being read into.

Read var[,var...]

Used to transfer items in Data statements into variables. Data is transferred sequentially into variables through what is known as a 'data pointer'. When a piece of data is read the data pointer is incremented to point at the next piece of data. Data pointer may be set to point to a particular piece of data using the Restore command.

Restore [label]

Allows you to set Blitz's internal 'data pointer' to a particular piece of data after executing a Restore. The first item of data following the specified label will become the data to be read when the next Read command is executed. Restore with no parameters will reset data pointer to very first piece of data in program.

Edit$([defaultstring$],characters)

This is Blitz's standard text input command. When used with Window and BitMap, Input Edit$ causes the optional defaultstring$ and a cursor to be printed to display. It then waits for the user to hit RETURN. Edit$ returns the text entered by program user as a string of characters.

During FileInput, Edit$ reads the next n characters from the open file or until the next endofline character (chr$(10)). To read data from files that is not standard ASCII (ignore EOL terminators) Inkey$ should be used instead of Edit$. Characters specifies a maximum number of allowable characters for input. This is extremely useful in preventing Edit$ from destroying display contents.

Edit([defaultvalue],characters)

This is Blitz's standard numeric input command. The same characteristics apply as those for Edit$ however, Edit of course only accepts numeric input.

lnkey$[(characters)]

Used to collect one or more characters from the current input channel. The current input channel may be selected using commands such as WindowInput, Filelnput or BitMapInput. Inkey$ MAY NOT be used from DefaultInput input channel as CLI does not pass input back to the program until the user hits return. Characters refers to the number of characters to collect. The default is one character.

DefaultInput

Causes all future Edit$ and Inkey$ functions to receive their input from CLI window the Blitz program was run from. This is the default input channel used when a Blitz program is first run.

DefaultOutput

Cause all future Print statements to send their output to CLI window the Blitz program was run from. This is the default output channel used when a Blitz program is first run.

FileRequest$(title$,pathname$,filename$)

This function will open up a standard Amiga style file requester on currently used screen. Program flow will halt until user either selects a file, or hits requester's 'Cancel' button. If a file was selected, FileRequest$ will return the full file name as a string. If 'Cancel' was selected, FileRequest$ will return a null (empty) string.

Title$ may be any string expression to be used as a title for the file requester.

The parameters for Pathname$ MUST be a string variable with a MaxLen of at least 160 and filename$ MUST be a string variable with a MaxLen of at least 64.

PopInput

After input has been re-directed (eg. using WindowInput/Filelnput), PopInput may be used to return the channel to its previous condition.

PopOutput

After output has been re-directed (eg using WindowOutput/FileOutput), PopOutput may be used to return the channel to its previous condition.

Joyx(port)

This will return the left/right status of a joystick plugged into the specified port. Port must be either 0 or 1 with 0 being the port the mouse is normally plugged into. If the joystick is held to the left, Joyx will return -1. If the joystick is held to the right, Joyx will return 1. If the joystick is held neither left or right, Joyx will return 0.

Joyy(port)

Will return the up/down status of a joystick plugged into the specified port. Port must be either 0 or 1 with 0 being the port the mouse is normally plugged into. If the joystick is held upwards, Joyy will return -1. If the joystick is held downwards, Joyy will return 1. If the joystick is held neither upwards or downwards, Joyy will return 0.

Joyr(port)

May be used to determine the rotational direction of a joystick plugged into the specified port. Port must be either 0 or 1 with 0 being the port the mouse is normally plugged into. Joyr returns a value from 0 through 8 based on the following table:

Direction	Value
Up	0
Up-Right	1
Right	2
Down-Right	3
Down	4
Down-Left	5
Left	6
Up-Left	7
No Direction	8

Joyb(port)

Read the button status of the device plugged into the specified port. Port must be either 0 or 1 with 0 being the port where mouse is normally plugged into. If the left button is pressed, Joyb will return 1. If right button is pressed, Joyb will return 2. If both buttons are pressed, Joyb will return 3. If no buttons are pressed, Joyb will return 0.

Gameb(port#)

Gameb returns the button states of CD32 style game controllers. The values of all buttons pressed are added together to make up the value returned by Gameb. To check if a certain button is down a logical AND should be performed, buttonvalue AND returnvalue will evaluate to 0 if the button is not held down. The button values are:

Button	Value
Play/Pause	1
Reverse	2
Forward	4
Green	8
Yellow	16
Red	32
Blue	64

R-4: FILE HANDLING & IFF INFO

Blitz supports 2 modes of file access: sequential and random. The following section covers the Blitz commands that open, close and operate these two types of files.

Blitz also contains special commands for finding information about ILBM files which are standard on the Amiga for containing graphics in the form of bitmaps and brushes. For specialised commands that read and write graphics and sound files more information and command descriptions are available in the appropriate sections.

OpenFile(file#,filename$)

Attempts to open the file specified by filename$. If the file was successfully opened, OpenFile will return true (-1), otherwise OpenFile will return false (0).

Files opened using OpenFile may be both written to and read from. If the file specified by filename$ did not already exist before the file was opened, it will be created.

Files opened with OpenFile are intended for use by the random access file commands although it is quite legal to use these files in a sequential manner.

ReadFile(file#,filename$)

Opens an existing file specified by filename$ for sequential reading. If the specified file was successfully opened, ReadFile will return true (-1), otherwise false (0) will be returned.

Once a file is open using ReadFile, FileInput may be used to read information from it.

WriteFile(file#,filename$)

Creates a new file specified by filename$ for the purpose of sequential file writing. If the file was successfully opened, WriteFile will return true (-1), otherwise false (0) will be returned.

A file opened using WriteFile may be written to by using the FileOutput command.

CloseFile file#

Used to close a file opened using one of the file open functions (FileOpen, ReadFile, WriteFile). This should be done to all files when they are no longer required.

Fields file#,var[,var...]

Set up fields of a random access file record. Once Fields is executed, Get and Put are used to read and write information to and from the file. The var parameters specify a list of variables you wish to be either read from or written to the file.

When a Put is executed the values held in these variables will be transferred to the file. When a Get is executed these variables will take on values read from the file.

Any string variables in the variable list MUST have been initialised to contain a maximum number of characters. This is done using the MaxLen command. These string variables must NEVER grow to be longer than their defined maximum length.

Put file#,record

Used to transfer the values contained in a Fields variable list to a particular record in a random access file. When using Put to increase the size of a random access file, you may only add to the immediate end of file. For example, if you have a random access file with 5 records in it, it is illegal to put record number 7 to the file until record number 6 has been created.

Get file#,record

Transfer information from a particular record of a random access file into a variable list set up by Fields command. Only records which also exist may retrieved.

FileOutput file#

Causes the output of all subsequent Print and NPrint commands to be sent to the specified sequential file. When the file is later closed, Print statements should be returned to an appropriate output channel (eg: DefaultOutput or WindowOutput).

FileInput file#

Causes all subsequent Edit, Edit$ and Inkey$ commands to receive their input from the specified file. When the file is later closed, input should be redirected to an appropriate channel (eg: DefaultInput or WindowInput).

FileSeek file#,position

Allows you to move to a particular point in the specified file. The first piece of data in a file is at position 0, the second at position 1 and so on. Position must not be set to a value greater than the length of the file.

Used in conjunction with OpenFile and Lof, FileSeek may be used to append to a file.

Lof(file#)

Returns the length, in bytes, of the specified file.

Eof(file#)

Allows you to determine if you are currently positioned at the end of the specified file. If so, Eof will return true (-1), otherwise Eof will return false (0). If you are at the end of a file, any further writing to the file will increase its length, while any further reading from the file will cause an error.

Loc(file#)

May be used to determine your current position in the specified file. When a file is first opened, you will be at position 0 in the file.

DosBuffLen bytes

All Blitz file handling is done through the use of special buffering routines. This is done to increase the speed of file handling, especially in the case of sequential files. Initially, each file opened is allocated a 2,048 byte buffer. However, if memory is tight this buffer size may be lowered using the DosBuffLen command.

KillFile filename$

Will simply attempt to delete the specified file. No error will be returned if the file could not be deleted.

CatchDosErrs

Whenever you are executing AmigaDOS I/O (for example, reading or writing a file), there is always the possibility of something going wrong (for example, disk not inserted... read/write error etc.). Normally, when such problems occur, AmigaDOS displays a suitable requester on the Workbench window. However, by executing CatchDosErrs you can force such requesters to open on a Blitz window.

The window you wish DOS error requesters to open on should be the currently used window at the time CatchDosErrs is executed.

ReadMem file#,address,length

Allows you to read a number of bytes, determined by length, into an absolute memory location determined by address from an open file specified by file#. Be careful using ReadMem, as writing to absolute memory may have serious consequences if you don't known what you're doing!

WriteMem file#,address,length

Allows you to write a number of bytes, determined by length from an absolute memory location determined by address to an open file specified by file#.

Exists(filename$)

Actually returns the length of the file. Unlike Lof(), Exists() is for files that have not already been opened. If 0 the file either doesn't exist, is empty or is perhaps not a file at all! Hmmm, anyway the following poke turns off the "Please Insert Volume Blah:" requester so you can use Exists to wait for disk changes:

```
Poke.l Peek.l(Peek.l(4)+276)+184,-1
```

ILBMInfo filename$

Examines an ILBM file. Once ILBMInfo has been executed, ILBMWidth, ILBMHeight and ILBMDepth examine properties of the image contained in file.

ILBMWidth

Returns the width (pixels) of an ILBM image examined with ILBMInfo.

ILBMHeight

Returns the height (pixels) of an ILBM image examined with ILBMInfo.

ILBMDepth

Returns the depth (bitplanes) of ILBM image examined with ILBMInfo.

ILBMViewMode

Returns the viewmode of the file that was processed by ILBMInfo. This is useful for opening a screen in the right mode before using LoadScreen etc. different values of ViewMode are as follows (add/or them for different combinations):

Mode	Value
HiRes	32768
Ham	2048
HalfBrite	128
Interlace	4
LoRes	0

R-5: NUMERIC & STRING FUNCTIONS

This section covers all Blitz functions which accept and return numeric and string values. Note that all the transcendental functions (eg. Sin, Cos) operate in radians.

Functions that return information about system time and date. Workbench parameters and so forth are also listed in this section.

True

True is a system constant with a value of -1.

False

False is a system constant with a value of 0.

NTSC

Returns 0 if the display is currently in PAL mode or -1 if currently in NTSC mode. This may be used to write software which dynamically adjusts itself to different versions of the Amiga computer.

DispHeight

Return 256 if executed on a PAL Amiga or 200 if on an NTSC Amiga. This allows programs to open full sized screens, windows,etc on any Amiga.

VPos

Returns the vertical position of the video beam. Useful in both high speed animation where the screen update may need to be synced to a certain video beam position (not just the top of frame as with VWait) and for a fast random member generator in non frame-synced applications.

Peek[.type](address)

Returns the contents of the absolute memory location specified by address. The optional type parameter allows peeking of different sizes. For example, to peek a byte, you would use Peek.b; to peek a word, you would use Peek.w; and to peek a long, you would use Peek.l. It is also possible to peek a string using Peek$. This will return a string of characters read from consecutive memory locations until a byte of 0 is found.

Abs(expression)

Returns the positive equivalent of expression.

Frac(expression)

Returns the fractional part of expression.

Int(expression)

This returns the integer part (before the decimal point) of expression.

QAbs(quick)

Works just like Abs except that the value it accepts is a quick. This enhances the speed at which the function executes quite dramatically. Of course you are limited by the restrictions of the quick type of value.

QFrac(quick)

Returns the fractional part of a quick value. It works like Frac() but accepts a quick value as its argument. It's faster than Frac() but has normal quick value limits.

QLimit(quick,low,high)

Used to limit the range of a quick number. If quick is greater than or equal to low, and less than or equal to high, the value of quick is returned. If quick is less than low, then low is returned. If quick is greater than high, then high is returned.

QWrap(quick,low,high)

Will wrap 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.

Rnd[(range)]

Returns a random number. If range isn't specified then a random decimal is returned between 0 and 1. If range is specified, then a decimal value between 0 and range is returned.

Sgn(expression)

Returns the sign of expression. If expression is less than 0, then -1 is returned. If expression=0 then 0 is returned. If expression is >0 then 1 is returned.

Cos(float)

Returns the cosine of the value float.

Sin(float)
Returns the sine of the value float.

Tan(float)
Returns the tangent of the value float.

ACos(float)
Returns the arc cosine of the value float.

ASin(float)
Returns the arc sine of the value float.

ATan(float)
Returns the arc tangent of the value float.

HCos(float)
Returns the hyperbolic cosine of the value float.

HSin(float)
Returns the hyperbolic sine of the value float.

HTan(float)
Returns the hyperbolic tangent of the value float.

Exp(float)
Returns e raised to the power of float.

Sqr(float)
Returns the square root of float.

Log10(float)
Returns the base 10 logarithm of float.

Log(float)
Returns the natural (base e) logarithm of float.

QAngle(srcx,srcy,destx,desty)

Returns the angle between the two 2D co-ordinates passed. The angle.q returned is a value from 0-1, 1 representing 360 degrees in standard polar geometry.

Left$(string$,length)

Returns the length leftmost characters of string string$.

Right$(string$,length)

Returns the rightmost length characters from string string$.

Mid$(string$,startchar[,length])

Returns length characters of string string$ starting at character startchar. If the optional length parameter is omitted, then all characters from startchar up to the end of string$ will be returned.

Hex$(expression)

Returns an 8 character string equivalent to hexadecimal representation of expression.

Bin$(expression)

Returns a 32 character string equivalent to a binary representation of expression.

Chr$(expression)

Returns a one character string equivalent to the ASCII character expression. ASCII is a standard way of coding the characters used by the computer display.

Asc(string$)

Returns the ASCII value of the first characters in the string string$.

String$(string$,repeats)

Return a string containing repeats sequential occurrences of the string string$.

Instr(string$,findstring$[,startpos])

Attempts to locate findstring$ within string$. If a match is found, it returns the character position of the first matching character. If no match is found, it returns 0. The optional startpos parameter allows you to specify a starting character position for the search.

CaseSense

Used to determine whether the search is case sensitive or not.

Replace$(string$,findstring$,replacestring$)

Search the string string$ for any occurrences of the string findstring$ and replace it with the string replacestring$. CaseSense is used to determine whether the search is case sensitive or not.

Mki$(integer)

Create a two byte character string, given the two byte numeric value numeric. Often used before writing integer values to sequential files to save disk space. When the file is later read in, Cvi may be used to convert the string back to an integer.

Mkl$(long)

Create a four byte character string, given the four byte numeric value long. Often used when writing long values to sequential files to save disk space. When the file is later read in, Cvl may be used to convert the string back to a long.

Mkq$(quick)

Create a four byte character string, given the four byte numeric value quick. Often used when writing quick values to sequential files to save disk space. When the file is later read in, Cvq may be used to convert the string back to a quick.

Cvi(string$)

Returns an integer value equivalent to the left 2 characters of string$. This is the logical opposite of Mki$.

Cvl(string$)

Returns a long value equivalent to the left 4 characters of string$. This is the logical opposite of Mkl$.

Cvq(string$)

Returns a quick value equivalent to the left 4 characters of string$. This is the logical opposite of Mkq$.

Len(string$)

Returns the length of the string string$.

UnLeft$(string$,length)

Removes the rightmost length characters from the string string$.

UnRight$(string$,length)

Removes the leftmost Length characters from the string string$.

StripLead$(string$,expression)

Removes all leading occurrences of the ASCII character specified by expression from the string string$.

StripTail$(string$,expression)

Removes all trailing occurrences of the ASCII character specified by expression from the string string$.

LSet$(string$,characters)

Returns a string of characters characters long. The string string$ will be placed at the beginning of this string. If string$ is shorter than characters the right hand side is padded with spaces. If it is longer, it will be truncated.

RSet$(string$,characters)

Returns a string of characters characters long. The string string$ will be placed at end of this string. If string$ is shorter than characters the left hand side is padded with spaces. If it is longer, it will be truncated.

Centre$(string$,characters)

Returns a string of characters characters long. The string string$ will be centred in the resulting string. If string$ is shorter than characters the left and right sides will be padded with spaces. If it is longer, it will be truncated on either side.

This function returns the string string$ converted into lowercase.

UCase$(string$)/LCase$(string$)

Returns the string string$ converted to uppercase/lowercase.

CaseSense On|Off

Allows control of the searching mode used by the Instr and Replace$ functions. CaseSense On indicates that an exact match must be found.

CaseSense Off indicates that alphabetic characters may be matched even if they are not in the same case. CaseSense On is the default search mode.

Val(string$)

Converts the string string$ into a numeric value and returns this value. When converting the string, the conversion will stop the moment either a non numeric value or a second decimal point is reached.

Str$(expression)

Returns a string equivalent of the numeric value expression. This now allows you to perform string operations on this string.

If the Format command has been used to alter numeric output, this will be applied to the resultant string.

UStr$(expression)

Returns a string equivalent of the numeric value expression. This now allows you to perform string operations on this string.

Unlike Str$, UStr$ is not affected by any active Format commands.

SystemDate

Returns the system date as the number of days passed since 1/1/1978.

Date$(days)

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).

NumDays(date$)

Converts a Date$ in the above format to the day count format, where numdays is the number of days since 1/1/1978.

DateFormat format# 0 or 1

Configures the way both date$ and numdays treat a string representation of the date: 0=dd/mm/yyyy and 1=mm/dd/yyyy

Days

Days, Months and Years each return the particular value relevant to the last call to SystemDate. They are most useful for when the program needs to format the output of the date other than that produced by date$. WeekDay returns which day of the week it is with Sunday=0 through to Saturday=6.

Months

See description of Days.

Years

See description of Days.

WeekDay

See description of Days.

Hours

Hours, Mins and Secs return the time of day when SystemDate was last called.

Mins

Hours, Mins and Secs return the time of day when SystemDate was last called.

Secs

Hours, Mins and Secs return the time of day when SystemDate was last called.

WBWidth

The functions WBWidth, WBHeight, WBDepth & WBViewMode return the width, height, depth & viewmode of the current Workbench screen as configured by preferences.

WBHeight

See Description of WBWidth.

WBDepth

See Description of WBWidth.

WBViewMode

See Description of WBWidth.

Processor

Returns the processor type in the computer on program is currently running.

0	= 68000
1	= 68010
2	= 68020
3	= 68030
4	= 68040

ExecVersion

Returns the relevant information about the system the program is running on.

33	= 1.2
34	= 1.3
36	= 2.0
39	= 3.0

R-6: COMPILER DIRECTIVES & OBJECT HANDLING

The following section refers to the Blitz Compiler Directives, commands which affect how a program is compiled. Conditional compiling, macros, include files and more are covered in this chapter.

Information regarding control of Blitz Objects is also listed in this section. Objects are Blitz's way of controlling specialised data concerned with windows and shapes etc.

USEPATH pathtext

Allows you to specify a 'shortcut' path when dealing with NEWTYPE variables. Consider the following lines of code:

```
aliens()\x= 160
aliens()\y= 100
aliens()\xs= 10
aliens()\ys=-10
```

USEPATH can be used to save you some typing, like so:

```
USEPATH aliens()
\x=160
\y=100
\xs=10
\ys=-10
```

Whenever Blitz encounters a variable starting with the backslash character ('\'), it simply inserts the current USEPATH text before the backslash.

BLITZ

Used to enter Blitz mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz Programmer's Guide.

AMIGA

Used to enter Amiga mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz Programmer's Guide.

QAMIGA

Used to enter Quick Amiga mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz Programmer's Guide.

INCLUDE filename

A compile-time directive which causes the specified file, filename, to be compiled as part of the programs object code. The file must be in tokenised form (eg. saved from the Blitz editor) - ASCII files may not be INCLUDE'd. INCDIR may be used to specify a path for filename.

Filename may be optionally quote enclosed to avoid tokenisation problems.

XINCLUDE filename

Exclusive include. XINCLUDE works identically to INCLUDE with the exception that XlNCLUDE'd files are only ever included once. For example, if a program has two XINCLUDE statements with the same filename, only the first XINCLUDE will have any effect.

IncBin filename

Allows you to include a binary file in your object code. This is mainly of use to assembler language programmers, as having big chunks of binary data in the middle of a BASIC program is not really a good idea. You may have to use the '?' character to reference a chunk of data that has been included. For example:

```
DecodeILBM 1,?img0
;more of the program
img0: IncBin "data/0.iff"
```

INCDIR pathname

May be used to specify a path for filename. Filename may be optionally quote enclosed to avoid tokenisation problems.

The INCDIR command allows you to specify an AmigaDOS path to be prefixed to any. Filenames specified by any of INCLUDE, XINCLUDE or INCBIN commands.

CNIF constant comparison constant

Allows you to conditionally compile a section of program code based on a comparison of two constants. Comparison should be one of '<', '>', '=', '<>', '<=' or '>='. If the comparison proves to be true, then compiling will continue. If comparison is false no object code will be generated until a matching CEND is encountered.

CEND

Marks the end of a block of conditionally compiled code. CEND must always appear somewhere following a CNIF or CSIF directive.

CSIF “string” comparison “string”

Allows you to conditionally compile a section of program code based on a comparison of two literal strings. Comparison should be one of ‘<’, ‘>’, ‘=’, ‘<>’, ‘<=’ or ‘>=’. Both strings must be quote enclosed literal strings. If the comparison proves to be true, then compiling will continue as normal. If the comparison proves to be false, then no object code will be generated until a matching CEND is encountered.

CSIF is of most use in macros for checking macro parameters.

CELSE

May be used between a CNIF or CSIF, and a CEND to cause code to be compiled when a constant comparison proves to be false.

CERR errormessage

Allows a program to generate compile-time error messages. CERR is normally used in conjunction with macros and conditional compiling to generate errors when incorrect macro parameters are encountered.

Macro macroname

Used to declare the start of a macro definition. All text following Macro, up until the next End Macro, will be included in the macro's contents.

End Macro

Used to finish a macro definition. Macro definitions are set up using the Macro command.

RunErrsOn

These two new compiler directives are for enabling and disabling error checking in different parts of the program, they override the settings in Compiler Options.

RunErrsOff

See description of RunErrsOn.

Use objectname object#

Will cause the Blitz object specified by Objectname and object# to become the currently used object.

Free Objectname object#

Frees a Blitz object. Any memory consumed by the object’s existence will be free’d up, and in case of things such as windows and screens, the display may be altered.

Attempting to free a non-existent object will have no effect.

USED objectname

Returns the currently used object number. Useful for routines which need to operate on currently used object, also interrupts should restore currently used object settings.

Addr objectname(object#)

A low-level function allowing advanced programmers the ability to find where a particular Blitz object resides in RAM. Appendix at the end lists all Blitz object formats.

Maximum objectname

Allows a program to determine the ‘maximum’ setting for a particular Blitz object. Maximum settings are entered into the OPTIONS requester, accessed through the ‘COMPILER’ menu of the Blitz editor.

R-7: ASSEMBLER DIRECTIVES

A powerful feature of Blitz is its built-in assembler. This allows the programmer to include machine code in their programs. You'll find the ability to mix easily BASIC with your own lightning fast machine code routines making a powerful connection.

There are three ways of including assembler in Blitz programs.

Inline: using PutRrg and GetReg BASIC variables can be exchanged with the 68000's data and address registers.

Procedures: Statements and Functions can contain 100% assembler, parameters are passed in registers D0...D5 and in case of Functions the value in D0 is returned to the caller. The AsmExit command is used in place of Statment Return or Function Return.

Libraries: Actual commands can be added to Blitz using assembler. See the libsdev archive in the blitzlibs: volume for more information.

Please note that when using assembler inline and within procedures address registers A4-A6 must be preserved. Blitz 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.

Also note that Absolute Short addressing mode and Short Branches are not supported.

DCB[.size] repeats,data

Stands for 'define consistent block'. DCB allows you to insert a repeating series of the same value into your assembler programs.

EVEN

Allows to word align Blitz's internal program counter. This may be necessary if a DC, DCB or DS statement has caused the program counter to be left at an odd address.

GetReg register,expression

Allows you to transfer the result of a BASIC expression to a 68000 register. The result of the expression will first be converted into a long value before being moved to the data register. Should only be used to transfer expressions to one of the 8 data registers (D0-D7). Will use the stack to temporarily store any registers used in calculation of the expression.

PutReg register,variable

May he used to transfer a value from any 68000 register (D0-D7/A0-A7) into a BASIC variable. If the specified variable is a string, long, float or quick, then all 4 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.

SysJsr routine

Allows you to call any of Blitz's system routines from your own program. Routine specifies a routine number to call.

TokeJsr token[,form]

Allows to call any of Blitz's library based routines. Token refers to either a token number, or an actual token name. Form refers to a particular form of the token.

ALibJsr token[,form]

Is only used when writing Blitz libraries. ALibJsr allows you to call a routine from another library from within your own library. Please refer to the Library Writing section of the programmer's guide for more information on library writing.

BLibJsr token[,form]

Is only used when writing Blitz libraries. BLibJsr allows you to call a routine from another library from within your own library. Please refer to the Library Writing section of the programmer's guide for more information on library writing.

AsmExit

Used to exit from functions and statements written in assembler. Registers A4-A6 must be preserved in functions and statements written in assembler.

R-8: MEMORY CONTROL

This section deals with low-level commands which allow you access to the Amiga's memory. Care must be taken when accessing memory in this way or an invitation to the alert guru may be mistakenly made.

Poke[.type] address,data

This command will place the specified data into a absolute memory location specified by address. The size of the Poke may be specified by the optional type parameter. For example, to poke a byte into memory use Poke.b; to poke a word into memory use Poke.w; and to poke a long word into memory use Poke.l

In addition, strings may be poked into memory by use of Poke$. This will cause the ASCII code of all characters in the string specified by data to be poked, byte by byte, into consecutive memory locations. An extra 0 is also poked past the end of the string.

Peek[.type](address)

Returns the contents of the absolute memory location specified by address. The optional type parameter allows peeking of different sizes. For example, to peek a byte, you would use Peek.b; to peek a word, you would use Peek.w; and to peek a long, you would use Peek.l

It is also possible to peek a string using Peek$. This will return a string of characters read from consecutive memory locations until a byte of 0 is found.

Peeks$(address,length)

Returns a string of characters corresponding to bytes peeked from consecutive memory locations starting at address and, length characters in length.

Call address

Call make program flow to be transferred to the memory location specified by address. NOTE that Call is for advanced programmers only, as incorrect use of Call can lead to severe problems - GURUS etc!

A 68000 JSR instruction is used to transfer program flow, so an RTS may be used to transfer back to the Blitz program.

Bank(bank#)

Returns the memory location of the given memory bank, replaces the older and more stupidly named BankLoc command.

BankSize(bank#)

Returns the size of the memory block allocated for the specified bank#.

lnitBank Bank#,size,memtype

Allocates a block of memory and assigns it to the bank specified. The memtype is the same as the Amiga operating system memory flags:

1	= public
2	= chip
65536	= clear memory

FreeBank bank#

De-allocates any memory block allocated tor the bank specified.

LoadBank bank#,filename$[,memtype]

This command has been modified. Instead of having to initialise the bank before loading a file, it will now initialise the bank to the size of the file if it is not already large enough or has not been initialised at all.

SaveBank bank#,filename$

Save the memory assigned to the bank to the filename specified.

AllocMem(size,type)

Unlike calling Exec's AllocMem_ command directly Blitz will automatically free any allocated memory when the program ends. Programmers are advised to use the InitBank command. Flags that can be used with the memory type parameter are:

1	= public	;fast if present
2	= chipmem	
65536	= clear	;clears all memory allocated with 0's

FreeMem location,size

Used to free any memory allocated with the AllocMem command.

R-9: PROGRAM STARTUP

This section covers all commands dealing with how an executable file goes about starting up. This includes the ability to allow your programs to run from Workbench and to pick up parameters supplied through the CLI.

WBStartup

By executing WBStartup at some point in your program, your program will be given the ability to run from Workbench. A program run from Workbench which does NOT include the WBStartup command will promptly crash if an attempt is made to run it from Workbench.

NumPars

Allows an executable file to determine how many parameters were passed to it by either Workbench or the CLI. Parameters passed from the CLI are typed following the program name and separated by spaces. For example. let's say you have created an executable program called myprog, and run it from the CLI in the following way:

```
1.SYS:> myprog filer 0lle2
```

In this case, NumPars would return the value 2 - ‘file1’ and ‘file2’ being the 2 parameters.

Programs run from Workbench are only capable of picking up 1 parameter through the use of either the parameter file’s ‘Default Tool’ entry in its ‘.info’ file, or by use of multiple selection through the ‘Shift’ key.

If no parameters are supplied to an executable file, NumPars will return 0. During program development, the ‘CLI Argument’ menu item in the ‘COMPILER’ menu allows you to test out CLI parameters.

Par$(parameter)

Returns a string equivalent to a parameter passed to an executable file through either the CLI or Workbench. Refer to NumPars for more information.

CloseEd

Causes the Blitz editor screen to ‘close down’ when programs are executed from within Blitz. This may be useful when writing programs which use a large amount of chip memory, as the editor screen itself occupies 40K of ChipMem. CloseEd will have no effect on executable files run outside of the Blitz environment.

NoCli

Prevents the normal ‘Default Cli’ from opening when programs are executed from within Blitz. NoCli has no effect on executable files run outside Blitz environment.

FromCLI

Returns TRUE (-1) if your program was run from CLI, or FALSE (0) if run from Workbench.

ParPath$(parameter,type)

Returns the path that the parameter resides in and ‘type’ specifies how you want the path returned:

0	You want only the directory of the parameter returned.
1	You want the directory along with the parameter name returned.

If you passed the parameter "FRED" to your program from Workbench, and FRED resides in the directory "work:mystuff/myprograms" then ParPath$(0,0) will return "work:mystuff/myprograms" but ParPath$(0,1) will return "work:mystuff/myprograms/FRED".

The way Workbench handles argument passing of directories is different to that of files. When a directory is passed as an argument, ArgsLib gets an empty string for the name, and the directory string holds the path to the passed directory AND the directory name itself.

R-10: SLICES

Slices are Blitz 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 more.

A slice may be thought of as a 'description' of the appearance of a rectangular area of the Amiga's display. This description includes display mode, colour palette, sprite and bitplane information. More than one slice may be set up at a time, allowing different areas of the display to take on different properties.

Slices must not overlap in any way (at least two Scan lines is required between each slice). They may not be positioned side by side.

Slice slice#,y,flags
Slice slice#,y,width,height,flags,bitplanes,sprites,colours,w1,w2

Used to create a Blitz slice object. Slices are primarily of use in Blitz mode, allowing you to create highly customized displays.

In both forms of the Slice command, the y parameter specifies the vertical pixel position of the top of the slice. A y value of 44 will position slices at about the top of the display.

In the first form of the Slice command, flags refers to the number of bitplanes in any bitmaps (the bitmap's depth) to be shown in the slice. This form of the Slice command will normally create a lo-res slice, however this may be changed to a hi-res slice by adding eight to the flags parameter. For instance, a flags value of four will set up a lo-res, 4 bitplane (16 colour) slice, whereas a flags value of ten will set up a hi-res, 2 bitplane (4 colour) slice. The width of a slice set up in this way will be 320 pixels for a lo-res slice, or 640 pixels for a hi-res slice. The height of a slice set up using this syntax will be 200 pixels on an NTSC Amiga, or 256 pixels on a PAL Amiga.

The second form of the Slice command is far more versatile, albeit a little more complex. Width and height allow you to use specific values for the slice's dimensions. These parameters are specified in pixel amounts.

Bitplanes refers to the depth of any bitmaps you will be showing in this slice. Sprites refers to how many sprite channels should be available in this slice. Each slice may have up to eight sprite channels, allowing sprites to be 'multiplexed'. This is one way to overcome the Amiga's 'eight sprite limit'. It is recommended that the top-most slice be created with all 8 sprite channels, as this will prevent sprite flicker caused by unused sprites.

Colours refers to how many colour palette entries should be available for this slice, and should not be greater than 32.

The w1 and w2 parameters specify the width, in pixels, of any bitmaps to be shown in this slice. If a slice is set up to be a dual-playfield slice, w1 refers to the width of the 'foreground' bitmap, and w2 refers to the width of the 'background' bitmap. If a slice is NOT set up to be a dual-playfield slice, both w1 and w2 should be set to the same value. These parameters allow you to show bitmaps which are wider than the slice, introducing the ability to smooth scroll through large bitmaps. The flags parameter has been left to last because it is the most complex.

Flags allows you control over many aspects of the slices appearance, and just what effect the slice has. Here are some example settings for flags:

Flags	Effect	Max BitPlanes
$fff8	A Standard lo-res slice	6
$fff9	A Standard hi-res slice	4
$fffa	A Lo-res, dual-playfield slice	6
$fffb	A Hi-res, dual-playfield slice	4
$fffc	A HAM slice	6

WARNING - the next bit is definitely for the more advanced users out there! Knowledge of the following is NOT necessary to make good use of slices. Flags is actually a collection of individual bit-flags. The bit-flags control how the slices 'copper list' is created. Here is a list of the bits and their effect:

Bit#	Effect
15	Create copper MOVE BPLCON0
14	Create copper MOVE BPLCON1
13	Create copper MOVE BPLCON2
12	Create copper MOVE DIWSTRT and MOVE DIWSTOP
10	Create copper MOVE DDFSTRT and MOVE DDFSTOP
8	Create copper MOVE BPL1MOD
7	Create copper MOVE BPL2MOD
4	Create a 2 line 'blank' above top of slice
3	Allow for smooth horizontal scrolling
2	HAM slice
1	Dual-playfield slice
0	Hi-res slice - default is lo-res

Clever selection of these bits allows you to create 'minimal' slices which may only affect specific system registers.

The bitplanes parameter may also be modified to specify ‘odd only’ or ‘even only’ bitplanes. This is of use when using dual playfield displays, as it allowing you to create a mid display slice which may show a different foreground or background bitmap leaving the other intact. To specify creation of foreground bitplanes only, simply set bit 15 of the bitplanes parameter. To specify creation of background bitplanes only, set bit 14 of the bitplanes parameter.

Use Slice slice#

Used to set the specified slice object as being the currently used slice. This is required tor commands such as Show, ShowF, ShowB and Blitz mode RGB.

FreeSlices

Used to free all slices currently in use. As there is no capability to free individual slices, this is the only means by which slices may be deleted.

Show bitmap#[,x,y]

Used to display a bitmap in the currently used slice. This slice should not be a dual-playfield type slice. Optional x and y parameters may be used to position the bitmap at a point other than its top-left. This is normally only of use in cases where a bitmap larger than the slice width and/or height has been set up.

ShowF bitmap#[,x,y[,ShowB x]]

Used to display a bitmap in the foreground of the currently used slice. The slice must have been created with the appropriate flags parameter in order to support dual-playfield display.

Optional x and y parameters may be used to show the bitmap at a point other than its top-left. Omitting the x and y parameters is identical to supplying values of 0. The optional ShowB x parameter is only of use in special situations where a dual-playfield slice has been created to display ONLY a foreground bitmap. In this case, the x offset of the background bitmap should be specified in the ShowB x parameter.

ShowB bitmap#[,x,y[,ShowF x]]

Used to display a bitmap in the background of the currently used slice. The slice must have been created with the appropriate flags parameter in order to support dual-playfield display.

Optional x and y parameters may be used to show the bitmap at a point other than its top-left. Omitting the x and y parameters is identical to values of 0. The optional ShowF x parameter is only of use in special situations where a dual-playfield slice has been created to display ONLY a background bitmap. In this case, the X offset of the foreground bitmap should be specified in the ShowF x parameter.

ColSplit colourregister,red,green,blue,y

Allows you to change any of the palette colour registers at a position relative to the top of the currently used slice. This allows you to 're-use' colour registers at different positions down the screen to display different colours. Y specifies a vertical offset from the top of the currently used slice.

CustomCop copin$,y

Allows advanced programmers to introduce their own copper instructions at a specified position down the display. Copins$ refers to a string of characters equivalent to a series of copper instructions. Y refers to a position down the display.

ShowBlitz

Redisplays the entire set up of slices. This may be necessary if you have made a quick trip into Amiga mode, and wish to return to Blitz mode with previously created slices intact.

CopLoc

Returns the memory address of the Blitz mode copper list. All Slices, ColSplits, and CustomCops executed are merged into a single copper list, the address of which may found using the CopLoc function.

CopLen

Returns the length, in bytes, of the Blitz mode copper list. All Slices, ColSplits, and CustomCops executed are merged into a single copper list, the length of which may found using the CopLen function.

Display On|Off

This is a Blitz mode only command which allows you to 'turn on' or 'turn off' the entire display. If the display is turned off, the display will appear as a solid block of colour 0.

SetBPLCON0 default

This command has been added for advanced control of Slice display modes. The bits of interest are as follows:

bit#1	ERSY external sync (for genlock enabling)
bit#2	LACE interlace mode
bit#3	LPEN light pen enable

The new display library is an alternative to the slice library. Instead of extending the slice library for AGA support a completely new display library has been developed.

Besides support for extended sprites, super hires scrolling and 8 bitplane displays 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 any combination of CopLists. Most importantly the CreateDisplay command does not allocate any memory avoiding any memory fragmenting problems. The new display library is for non-AGA displays also.

To create displays the InitCopList command is used to allocate memory for what were up till now known as Slices. A display is then created by linking one or more of these coplists together into a single display.

With many of the new AGA modes sprite DMA has been screwed up something severe. Those wanting to use 8 bitplanes and 8 sprites in lores will be disappointed to hear that their displays must be modified to some 256 pixels across.

The way the Amiga fetches data for each scan line is also a little different with the AGA machines. The effect is that displays have to be created more to the right than usual so

the system has time to fetch sprites.

InitCopList coplist#,ypos,height,type,sprites,colors,customs

Used to create a CopList for use with the CreateDisplay command. The ypos and height parameters define the vertical section of the screen the display will take up.

Sprites, colors and customs will allocate instructions for that many sprites (always=8!) colours (yes, as many as 256!) and custom copper instructions (which need to be allocated to take advantage of the custom commands listed at the end of this section).

A shortened version of the InitCopList command is available that simply requires the CopList# and the type. From the type it fills in the missing parameters. As with slices several lines must be left between coplists when displaying more than one.

The following constants make up the type parameter, add the number of bitplanes to the total to make up the type parameter:

Type	Value
#smoothscroll	$0010
#dualplayfield	$0020
#extrahalfbrite	$0040
#ham	$0080
#lores	$0000

#hires	$0100
#super	$0200
#loressprites	$0400
#hiressprites	$0800
#supersprites	$0c00
#fmode0	$0000
#fmode1	$1000
#fmode2	$2000
#fmode3	$3000
#agapalette	$10000

For displays on non-AGA machines only #fmode0 and #loressprites are allowed. More documentation, examples and fixes will be published soon for creating displays.

CreateDisplay coplist#[,coplist#...]

Used to setup 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.

DisplayBitMap coplist#,bmap[,x,y][,bmap[,x,y]]

This command is similar in usage to the slice libraries' show commands instead of different commands for front and back playfields and smooth scroll options there is only the one DisplayBitMap command with various parameter options With AGA machines, the x positioning of lores and hires coplists uses the fractional part of the x parameter for super smooth scrolling. The coplist must be initialised with the smooth scrolling flag set if the x,y parameters are used, same goes to dualplayfield.

DisplaySprite coplist#,sprite#,x,y,spritechannel

This is similar to the slice libraries ShowSprite command with the added advantage of super hires positioning and extra wide sprite handling. See also SpriteMode and the usage discussion above.

DisplayPalette coplist#,palette#[,coloroffset]

Copies colour information from a palette to the coplist specified.

DisplayControls coplist#,BPLCON2,BPLCON3,BPLCON4

Allows access to the more remote options available in the Amiga's display system. The following are the most important bits from these registers (still unpublished by Commodore!*()@GYU&^)

Default values are at top of the table, parameters are exclusive OR'd with these values. To set all the sprite color offsets to 1 so that sprite colours are fetched from color registers 240…255 instead of 16…31 we would use the parameters:

```
DisplayControls 0,0,0,$ee
```

Bit#	BPLCON2	BPLCON3	BPLCON4
	($224)	($c00)	($11)
15	*	BANK2 *active colour bank	BPLAM7 ;xor with bitplane
14	ZDBPSEL2	BANK1 *	BPLAM6 ;DMA altering
13	ZDBPSEL1	BANK0 *	BPLAM5 ;effective colour
12	ZDBPSEL0	PF20F2 colourffset playfield 2	BPLAM4 ;look up
11	ZDBPEN	PF20F1	BPLAM3
10	ZDCTEN	PF20F0	BPLAM2
09	KILLEHB *	LOCT *palette hi/lo nibble	BPLAM1
08	RDRAM=0 *		BPLAM0
07	SOGEN	SPRESI *sprite res	ESPRM7 high order colour
06	PF2PRI H	SPRES0 *	ESPRM6 offset tor even
05	PF2P2	BRDRBLANK border	ESPRM5 sprites
04	PF2P1	BRDNTRAN zd=border	ESPRM4
03	PFIP0		OSPRM7 hiorder colour
02	PFIP2	ZDCLCKEN zd=14mhz	OSPRM6 offset for odd
01	PFIPI	BRDSPRT sprites in borders!	OSPRM5 sprites
00	PFIPO	EXTBLKEN blank output?	OSPRM4

!	= Don't touch
H	= See standard hardware reference manual
*	= controlled by display library
ZD	= any reference to ZD is only a guess (just sold my genlock)

DisplayAdjust coplist#,fetchwid,ddfstrt,ddfstop,diwstrt,diwstop

Temporary control of display registers until I get the width adjust parameter working with InitCopList. Currently only standard width displays are available but you can modify the width manually (just stick a screwdriver in the back of your 1084) or with some knowledge of Commodore's AGA circuitry. Ha ha ha! No, to be quite serious I really do not have a clue how they cludged up the Amiga chip set. When ECS was introduced suddenly all display fetching moved to the right. Now they seem to have done the same to sprites so it is near impossible to have them all going without limiting yourself to a seriously thin display.

If you hack around with the system copperlists you'll find they actually change fetch modes as you scroll a viewport across the display and Commodore say you should not use sprites anyway so as to be compatible with their new hardware which is rumoured to run WindowsNT, yipeee. By then we will be hopefully shipping the Jaguar lib for Blitz.

CustomColors coplist#,ccoffset,ypos,palette,startcol,numcols

Using the custom copper space in a display, CustomColors will alter the displays palette at the given ypos. The number of custom cops required is either 2+numcols for ECS displays and 2+n+n+n/16 for AGA displays. In AGA, numcols must be a multiple of 32.

Note: Large AGA palette changes may take several lines of the display to be complete.

CustomString coplist#,ccoffset,ypos,copper$

Allows the user to insert their own copper commands (contained in a string) into the display's copper list at a given vertical position. The amount of space required is equal to the number of copper instructions in copper$ (length of string divide by 4) plus 2 which of course have to be allocated with InitCopList before CustomString is used.

CustomSprites coplist#,ccoffset,ypos,numsprites

Inserts a copper list that reinitialises the sprites hardware at a certain vertical position in the display. These lower sprites are assigned sprite numbers of 8...15. CustomCops required = 4 x numsprites + 2

DisplayDblScan mode

Used to divide the vertical resolution of the display by 2,4,8 or 16 using Modes 1,2,3 and 4. This is most useful for fast bitmap based zooms. A mode of 0 will return the display to 100% magnification.

As with the DisplayRainbow, DisplayRGB, DisplayUser and DisplayScroll commands DisplayDblScan uses the new line by line copper control of the display library. To initialise this mode a negative parameter is used in the CustomCops parameter of the InitCopList command. DisplayDblScan requires 2 copper instructions per line (make CustomCops=-2).

DisplayRainbow coplist#,register,palette[,copoffset]

Used to alter a certain colour register vertically down a display. It simple maps each colour in a palette to the corresponding vertical position of display. ECS displays require one copper instruction per line while AGA displays require 4.

DisplayRGB coplist#,register,line,r,g,b[,copoffset]

This is a single line version of DisplayRainbow allowing the programmer to alter any register of any particular line. As with DisplayRainbow ECS displays require 1 copper instruction while AGA requires 4.

DisplayUser coplist#,line,string[,offset]

Allows the programmer to use their own copper$ at any line of the display. Of course copper instructions have to be allocated with the number of copper instructions in the InitCoplist multiplied by -1.

DisplayScroll coplist#,&xpos.q(n),&xpos.q(n)[,offset]

Allows the program to dynamically display any part of a bitmap on any line of the display. DisplayScroll should always follow the DisplayBitMap command. The parameters are two arrays holding a list of xoffsets that represent the difference in horizontal position from the line above. AGA machines are able to use the fractional part of each entry for super hi resolution positioning of the bitmap. Three instructions per line are required for the DisplayScroll command.

R-12: BLITZ MODE I/O

This sections refers to various Input/Output commands available in Blitz mode.

It should be noted that although the Joyx, Joyy, Joyr, and Joyb functions do not appear here, they are still available in Blitz mode (yes your honour).

BlitzKeys On|Off

Used to turn on or off Blitz mode keyboard reading. If Blitz mode keyboard reading is enabled, the Inkey$ function may be used to gain information about keystrokes in Blitz mode.

BlitzQualifier

Returns any qualifier keys that were held down in combination with the last Inkey$ during BlitzMode input.

BlitzRepeat delay,speed

Allows you to determine key repeat characteristics in Blitz mode. Delay specifies the amount of time, in fiftieths of a second, before a key will start repeating. Speed specifies the amount of time, again in fiftieths of a second, between repeats of a key once it has started repeating.

BlitzRepeat is only effective when the Blitz mode keyboard reading is enabled. This is done using the BlitzKeys command.

RawStatus (Rawkey)

This function can be used to determine if an individual key is being held down or not. Rawkey is the rawcode of the key to check for. If the specified key is being held down, a value of -1 will be returned. If the specified key is not being held down, a value of zero will be returned.

RawStatus is only available if Blitz mode keyboard reading has been enabled. This is done using the BlitzKeys command.

Mouse On|Off

Turns on or off Blitz mode's ability to read the mouse. Once a Mouse On command has been executed, programs can read the mouse's position or speed in Blitz mode.

Pointer sprite#,spritechannel

Allows you to attach a sprite object to the mouse's position in the currently used slice in Blitz mode.

To properly attach a sprite to mouse position, several commands must be executed in the correct sequence. First, a sprite must be created using the LoadShape and GetaSprite sequence of commands. Then, a slice must be created to display the sprite in. A Mouse On must then be executed to enable mouse reading.

MouseArea minx,miny,maxx,maxy

Allows you to limit Blitz mode mouse movement to a rectangular section of the display. Minx and miny define the top left corner of the area, maxx and maxy define the lower right corner.

MouseArea defaults to an area from 0,0 to 320,200.

MouseX

If Blitz mode mouse reading has been enabled using a Mouse On command, the MouseX function may be used to find the current horizontal location of the mouse. If mouse reading is enabled, the mouse position will be updated every fiftieth of a second, regardless of whether or not a mouse pointer sprite is attached.

MouseY

If Blitz mode mouse reading has been enabled using Mouse On command, the MouseY function may be used to find the current vertical location of the mouse. If mouse reading is enabled, the mouse position will be updated every fiftieth of a second, regardless of whether or not a mouse pointer sprite is attached.

MouseXSpeed

If Blitz mode mouse reading has been enabled using a Mouse On command, the MouseXSpeed function may be used to find the current horizontal speed of mouse movement, regardless of whether or not a sprite is attached to the mouse.

If MouseXSpeed returns a negative value, then the mouse has been moved to the left. If a positive value is returned, the mouse has been moved to the right. MouseXSpeed only has relevance after every vertical blank. Therefore, MouseXSpeed should only be used after a VWait has been executed or during a vertical blank interrupt.

MouseYSpeed

If Blitz mode mouse reading has been enabled using a Mouse On command, the MouseYSpeed function may be used to find the current vertical speed of mouse movement, regardless of whether or not a sprite is attached to the mouse.

If MouseYSpeed returns a negative value, then the mouse has been moved upwards. If a positive value is returned, the mouse has been moved downwards.

MouseYSpeed only has relevance after every vertical blank. Therefore, MouseYSpeed should only be used after a VWait has been executed or during a vertical blank interrupt.

LoadBlitzFont blitzfont#,fontname.font$

Creates a blitzfont object. Blitzfonts are used in the rendering of text to bitmaps. Normally, the standard ROM resident Topaz font is used to render text to bitmaps. However, you may use LocalBlitzFont to select a font of your choice for bitmap output.

The specified Fontname.font$ parameter specifies the name of the font to load, which MUST be in your FONTS: directory.

LoadBlitzFont may only be used to load 8x8 non-proportional fonts.

Use BlitzFont blitzfont#

If you have loaded two or more blitzfont objects using LoadBlitzFont, UseBlitzFont may be used to select one of these fonts for future bitmap output.

Free BlitzFont blitzfont#

This ‘unloads’ a previously loaded blitzfont object. This frees up any memory occupied by the font.

BitMapOutput bitmap#

May be used to redirect Print statements to be rendered onto a bitmap. The font used for rendering may be altered using LoadBlitzFont.

Fonts used for bitmap output must be 8x8 non-proportional fonts. BitMapOutput is mainly of use in Blitz mode as other forms of character output become unavailable in Blitz mode.

Colour fgcolour[,bgcolour]

Allows you to alter the colours use to render text to bitmaps. The parameter fgcolour allows you to specify the colour text is rendered in, and the optional bgcolour parameter allows you to specify the colour of the text background.

The palette used to access these colours will depend upon whether you are in Blitz mode or in Amiga mode. In Blitz mode, colours will come from the palette of the currently used slice. In Amiga mode, colours will come from the palette of the screen the bitmap is attached to.

Locate x,y

If you are using BitMapOutput to render text, Locate allows you to specify the cursor position at which characters are rendered.

X specifies a character position across the bitmap, and is always rounded down to a multiple of an eighth.

Y specifies a character position down the bitmap, and may be a fractional value. For example, a Y of 1.5 will set a cursor position one and a half characters down from the top of the bitmap.

CursX

When using BitMapOutput to render text to a bitmap, CursX may be used to find the horizontal character position at which the next character Printed will appear. CursX will reflect the cursor position of the bitmap specified in the most recently executed BitMapOutput statement.

CursY

When using BitMapOutput to render text to a bitmap, CursY may be used to find the vertical character position at which the next character Printed will appear. CursY will reflect the cursor position of the bitmap specified in the most recently executed BitMapOutput statement.

BitMapInput

This is a special command designed to allow you to use Edit$ and Edit in Blitz mode. To work properly, a BlitzKeys On must have been executed before BitMapInput. BitMapOutput must be executed before any Edit$ or Edit commands are encountered.

R-13: BITMAPS

Blitz bitmap objects are used primarily for the purpose of rendering graphics. Most commands in Blitz for generating graphics (excluding the Window and Sprite commands) depend upon a currently used bitmap.

Bitmap objects may be created in one of two ways. A bitmap may be created by using the BitMap command, or a bitmap may be ‘borrowed’ from a screen using the ScreensBitMap command.

Bitmaps have three main properties. They have a width, a height and a depth. If a bitmap is created using the ScreensBitMap command, these properties are taken from the dimensions of the screen. If a bitmap is created using the BitMap command, these properties must be specified.

BitMap bitmap#,width,height,depth

Creates and initialises a bitmap object. Once created, the specified bitmap becomes the currently used bitmap. Width and height specify the size of the bitmap. Depth specifies how many colours may be drawn onto the bitmap, and may be in the range one through six. The actual colours available on a bitmap can be calculated using 2^depth. For example, a bitmap of depth three allows for 2^3 or eight colours.

Use BitMap bitmap#

Defines the specified bitmap object as being the currently used bitmap. This is necessary for commands, such as Blit, which require the presence of a currently used bitmap.

Free BitMap bitmap#

Erases all information connected to the specified bitmap. Any memory occupied by the bitmap is also deallocated. Once free’d, a bitmap may no longer be used.

CopyBitMap bitmap#,bitmap#

Makes an exact copy of a bitmap object into another bitmap object. The first bitmap# parameter specifies the source bitmap for the copy, the second bitmap# the destination.

Any graphics rendered onto the source bitmap will also be copied.

ScreensBitMap screen#,bitmap#

Allows you the option of attaching a bitmap object to any Intuition screens you open. If you open a screen without attaching a bitmap, a bitmap will be created anyway. You may then find this bitmap using the ScreensBitMap command. Once ScreensBitMap is executed, the specified bitmap becomes the currently used bitmap.

LoadBitMap bitmap#,filename$[,palette#]

Allows you to load an ILBM IFF graphic into a previously initialised bitmap object. You may optionally load in the graphics's colour palette into a palette object specified by palette#. An error will be generated if the specified filename$ is not in the correct IFF format.

SaveBitmap bitmap#,filename$[,palette#]

Allows you to save a bitmap to disk in ILBM IFF format. An optional palette may also be saved with the IFF.

BitPlanesBitMap srcbitmap,destbitmap,planepick

Creates a 'dummy' bitmap from the srcbitmap with only the bitplanes specified by the planepick mask. This is useful for shadow effects etc. where blitting speed can be speed up because of the fewer bitplanes involved.

ShapesBitMap shape#,bitmap#

Creates a dummy bitmap so drawing commands can be used directly on a shape's image data.

CludgeBitMap bitmap#,width,height,depth,memory

Creates a bitmap object with the proportions for that specified using the memory location given. Of course, the memory location specified must be in chipmem and it is upto the user to ensure that sufficient memory has been allocated. This command is most useful for games where memory fragmentation can be a big problem, by allocating one block of memory on program initialisation for all bitmaps CludgeBitMap can be used so that creating and freeing of bitmaps is not necessary.

BitMapWindow srcbitmap#,destbitmap#,x,y,w,h

Creates a dummy bitmap inside another bitmap. Both x and w parameters are rounded to the nearest 16 pixel boundary. Any rendering, printing and blitting to the new bitmap will be clipped inside the area used.

BitMapOrigin bitmap#,x,y

Allows the programmer to relocate the origin (0,0) of the bitmap used by the drawing commands line, poly, box and circle.

DecodeILBM bitmap#,sourceaddr

A very fast method of unpacking standard IFF ILBM data to a bitmap. Not only does this command allow a faster method of loading standard IFF files but allows the programmer to "IncBin" IFF pictures in their programs. See the discussion above for using DecodeILBM on both files and included memory.

If you do plan on using this command to include images in your program, you'll need to reference the label with the '?' character, like this:

```
DecodeILBM 0,?img0
;more of the program
img0: IncBin "data/0.iff"
```

The image will be unpacked at thje top-left (0,0) of your bitmap. If you want to position it, you'll first need to grab it and you can then blit it wherever you want inside a bitmap:

```
BitMap 1,14,24,5        ;Create a bitmap the same size as our IFF file
DecodeILBM 1,?img0      ;Decode the data to a bitmap
Blit 0,50,50            ;Blit it at 50,50 on the screen (bitmap 0)
img0: IncBin "data/0.iff"
```

Note: You can also use ILBMGrab command which also extracts the palette.

ILBMGrab sourceaddr,bitmap#,palette#

Works in much the same way as DecodeILBM except this will also extract the palette of the IFF ILBM file as well.

```
BitMap 0,320,256,5
Screen 0,0,0,320,256,5,0,"Picture Test",0,1
ILBMGrab ?tigerpic,0,0
ShowBitMap 0
ShowPalette 0
MouseWait
End
titlepic: IncBin "data/tiger01.iff"
```

R-14: 2D DRAWING

This section covers all commands related to rendering arbitrary graphics to bitmaps All commands perform clipping - that is, they all allow you to draw 'outside' the edges of bitmaps without grievous bodily harm being done to the Amiga's memory.

Cls [colour]

Allows you to fill the currently used bitmap with the colour specified by the colour parameter. If colour is omitted, the currently used bitmap will be filled with colour 0. A colour parameter of -1 will cause the entire bitmap to be 'inverted'.

Plot x,y,colour

Used to alter the colour of an individual pixel on the currently used bitmap. The parameters x and y specify the location of the pixel to be altered and colour specifies the colour to change the pixel to. A parameter of -1 for colour will cause the pixel at the specified pixel position to be 'inverted'.

Point(x,y)

Return the colour of a particular pixel in the currently used bitmap. The pixel to be examined is specified by the x and y parameters If x and y specify a point outside the edges of the bitmap, a value of -1 will be returned.

Line [x1,y1,]x2,y2,colour

Draws a line connecting two pixels onto the currently used bitmap. The x and y parameters specify the pixels to be joined and colour specifies the colour to draw the line in. If x1 and y1 are omitted then the end points (x2,y2) of the last line drawn will be used. A colour parameter of -1 will cause an 'inverted' line to be drawn.

Box x1,y1,x2,y2,colour

Draws a rectangular outline onto the currently used bitmap. Paremeters x1, y1, x2 and y2 specify two corners of the box to be drawn. Colour refers to the colour to draw the box in. A colour parameter of -1 will cause an 'inverted' box to be drawn.

Boxf x1,y1,x2,y2,colour

Draws a solid rectangular shape on the currently used bitmap. Paremeters x1,y1,x2 and y2 refer to two corners of the box. Colour specifies the colour to draw the box in. A colour parameter of -1 will cause the rectangular area to be 'inverted'.

Circle x,y,radius[,yradius],colour

Draws an open circle onto the currently used bitmap. Paremeters x and y specify the mid point of the circle. The radius parameter specifies the radius of the circle. If a yradius parameter is supplied, then an ellipse may be drawn. A colour parameter of -1 will cause an ‘inverted’ circle to be drawn.

Circlef x,y,radius[,yradius],colour

Circlef will draw a filled circle onto the currently used bitmap. Parameters x and y specify the mid point of the circle and colour, the colour in which to draw the circle. The radius parameter specifies the radius of the circle. If a yradius parameter is supplied, then an ellipse may be drawn.

A colour parameter of - 1 will cause an ‘inverted’ circle to be drawn.

Scroll x1,y1,width,height,x2,y2[,srcbitmap]

Allows rectangular areas within a bitmap to be moved around. Paremeters x1, y1, width and height specify the position and size of the rectangle to be moved. Paremeters x2 and y2 specify the position the rectangle is to be moved to.

An optional srcbitmap parameter allows you to move rectangular areas from one bitmap to another.

FloodFill x,y,colour[,bordercolour]

Will ‘colour in’ a region of the screen starting at the coordinates x,y. The first mode will fill all the region that is currently the colour at the coordinates x,y with the colour specified by colour. The second mode will fill a region starting at x,y and surrounded by the bordercolour with colour.

FreeFill

Deallocates the memory that Blitz uses to execute the commands Circlef, FloodFill, ReMap and Boxf.

Blitz uses a single monochrome bitmap the size of the bitmap being drawn to do its filled routines, by using the FreeFill command this bitmap can be ‘freed’ up if no more filled commands are to be executed.

ReMap colour#0,colour#1[,bitmap]

Used to change all the pixels on a bitmap in one colour to another colour. The optional bitmap parameter will copy all the pixels in colour#0 to their new colour on the new bitmap.

Poly numpoints,*coords.w,colour

Poly is a bitmap based commands such as Box and Line. It draws a polygon using coordinates from an array or newtype of words.

Polyf numpoints,*coords.w,color[,color2]

Same as Poly except Polyf draws filled polygons and has an optional parameter color2. If used this colour will be used if the coordinates are listed in anti-clockwise order, useful for 3D type applications. If color2=-1 then the polygon is not drawn if the vertices are listed in anti-clockwise order.

R-15: ANIMATION SUPPORT

The following four commands allow the display of standard IFF animations in Blitz. The animation must be compatible with the DPaint 3 format, this method uses long delta (type 2) compression and does not include any palette changes.

Anims in nature use a double buffered display, with the addition of the ShowBitMap command to Blitz we can now display (play) anims in both Blitz and Amiga modes. An anim consists of an initial frame which needs to be displayed (rendered) using the InitAnim command, subsequent frames are then played by using the NextFrame command. The Frames() function returns the number of frames of an anim.

We have also extended the LoadShape command to support anim brushes.

LoadAnim anim#,filename$[,palette#]

This will create an anim object and load a DPaint compatible animation. The ILBMInfo command can be used to find the correct screen size and resolution for the anim file. The optional palette# parameter can be used to load a palette with the anims correct colours.

InitAnim anim#[,bitmap#]

Renders the first two frames of the anim onto the current bitmap and the bitmap specified by the second parameter. The second bitmap# parameter is optional, this is to support anims that are not in a double-buffered format (each frame is a delta of the last frame not from two frames ago). However, the two parameter double buffered form of InitAnim should always be used. (hmmm don't ask me O.K.!)

NextFrame anim#

Renders the next frame of an anim to the current bitmap. If the last frame of an anim has been rendered NextFrame will loop back to the start of the animation.

Frames(anim#)

Returns the number of frames in the specified anim.

R-16: SHAPE HANDLING

Shape objects are used for the purpose of storing graphic images. These images may be used in a variety of ways. For example, a shape may be used as the graphics for a gadget, or as the graphics for a menu item or perhaps an alien being bent on your destruction.

See the Blitting section for the many commands that are available for the purpose of drawing shapes onto bitmaps. These commands use the Amiga's blitter chip to achieve this, and are therefore very fast.

Note that Blitz supports two different file formats for storage of shapes. Standard IFF brush files (such as created with DPaint) as well as anim brushes use the LoadShape/SaveShape commands and the faster Blitz format uses the LoadShapes and SaveShapes format.

LoadShape shape#,filename$[,palette#]

Allows you to load an ILBM IFF file into a shape object. The optional palette# parameter lets you also load the colour information contained in the file into a palette object.

This command has now been extended to support anim brushes. If the file is an anim brush the shapes are loaded into consecutive shapes starting with the shape# provided.

SaveShape shape#,filename$,palette#

Creates an ILBM IFF file based on the specified shape object. If you want the file to contain colour information, you should also specify a palette object using the palette# parameter.

LoadShapes shape#[,shape#],filename$

Lets you load a 'range' of shapes from disk into a series of shape objects. The file specified by filename$ should have been created using SaveShapes command.

The first shape# parameter specifies the number of the first shape object to be loaded. Further shapes will be loaded into increasingly higher shape objects.

If a second shape# parameter is supplied, then only shapes up to and including the second shape# value will be loaded. If there are not enough shapes in the file to fill this range, any excess shapes will remain untouched.

SaveShapes shape#,shape#,filename$

Allows you to create a file containing a range of shape objects. This file may be later loaded using the LoadShapes command.

The range of shapes to be saved is specified by shape#,shape#, where the first shape# refers to the lowest shape to be saved and the second shape# the highest.

GetaShape shape#,x,y,width,height

Lets you transfer a rectangular area of the currently used bitmap into the specified shape object. Paremeters x, y, width and height specify the area of the bitmap to be picked up and used as a shape.

CopyShape shape#,shape#

Produces an exact copy of one shape object in another shape object. The first shape# specifies the source shape for the copy, the second specifies the destination shape.

CopyShape is often used when you require two copies of a shape in order to manipulate (using, for example, XFlip) one of them.

AutoCookie On|Off

When shapes objects are used by any of the blitting routines (for example Blit), they usually require the presence of what is known as a 'cookiecut'. These cookiecuts are used for internal purposes by the various blitting commands, and in no way affect the appearance or properties of a shape. They consume some of your valuable chip mem.

When a shape is created (for example, by using LoadShape or GetaShape), a cookiecut is automatically made for it. However, this feature may be turned off by executing an AutoCookie Off command.

This is a good idea if you are not going to be using shapes for blitting - for example, shapes used for gadgets or menus.

MakeCookie shape#

Allows you to create a 'cookiecut' for an individual shape. Cookiecuts are necessary for shapes which are to be used by the various blitting commands (for example QBlit), and are normally made automatically whenever a shape is created (for example using LoadShape). However, use of the AutoCookie command may mean you end up with a shape which has no cookiecut, but which you wish to blit at some stage.

You can then use MakeCookie to make a cookiecut for this shape.

ShapeWidth(shape#)

Returns the width, in pixels, of a previously created shape object.

ShapeHeight(shape#)

Returns the height, in pixels,of a previously created shape object.

Handle shape#,x,y

All shapes have an associated 'handle'. A shape's handle refers to an offset from the upper left of the shape to be used when calculating a shapes position when it gets blitted to a bitmap. This is also often referred to as a 'hot spot'.

The x parameter specifies the horizontal offset for a handle, the y parameter specifies a vertical offset.

Let's have a look at an example of how a handle works. Assume you have set a shape's x handle to 5, and its y handle to 10. Now let's say we blit the shape onto a bitmap at pixel position 160,100. The handle will cause the upper left corner of the shape to end up at 155,90, while the point within the shape at 5,10 will end up at 160,100.

When a shape is created, its handle is automatically set to 0,0 - its upper left corner.

MidHandle shape#

Causes the handle of the specified shape to be set to its centre. For example, these two commands achieve exactly the same result:

```
MidHandle 0
Handle 0,ShapeWidth(0)/2,ShapeHeight(0)/2
```

For more information on handles, please refer to the Handle command.

XFlip shape#

One of Blitz's powerful shape manipulation commands. XFlip will horizontally 'mirror' a shape object, causing the object to be 'turned back to front'.

YFlip shape#

Used to vertically 'mirror' a shape object. The resultant shape will appear to have been 'turned upside down'.

Scale shape#,xratio,yratio[,palette#]

This is a very powerful command which may be used to 'stretch' or 'shrink' shape objects. The xratio and yratio parameters specify how much stretching or shrinking to perform. A ratio greater than one will cause the shape to be stretched (enlarged), while a ratio of less than one will cause the shape to be shrunk (reduced). A ratio of exactly one will cause no change in the shape's relevant dimension.

As there are separate ratio parameters for both x and y, a shape may be stretched along one axis and shrunk along the other!

The optional Palette# parameter allows you to specify a palette object for use in the scaling operation. If a Palette# is supplied, the scale command will use a ‘brightest pixel’ method of shrinking. This means a shape may be shrunk to a small size without detail being lost.

Rotate shape#,angleratio

Allows you to rotate a shape object. The angleratio specifies how much clockwise rotation to apply, and should be in the range zero to one. For instance, a value of .5 will cause a shape to be rotated 180 degrees, while a value of .25 will cause a shape to be rotated 90 degrees clockwise.

DecodeShapes shape#[,shape#],memorylocation

Similar to DecodeMedModule, ensures the data is in chip and then configures the shape object(s) to point to the data.

InitShape shape#,width,height,depth

Has been added to simple create blank shape objects. Programmers who make a habit of using ShapesBitMap to render graphics to a shape object will appreciate this one for sure.

R-17: BLITTING

The process of putting a shape onto a bitmap using the blitter is often referred to as 'blitting' a shape. The speed at which a shape is blitted is important when you are writing animation routines, as the smoothness of any animation will be directly affected by how long it takes to draw the shapes involved in the animation.

The two main factors which affect the speed at which a shape is blitted are its size and the technique used to actually blit the shape.

This section will cover all commands which allow you to draw shapes onto bitmaps using the Amiga's 'blitter' chip.

Blit shape#,x,y[,excessonoff]

The simplest of all the blitting commands. Blit will simply draw a shape object onto the currently used bitmap at the pixel position specified by x,y. The shape's handle, if any, will be taken into account when positioning the blit.

The optional excessonoff parameter only comes into use if you are blitting a shape which has less bitplanes (colours) than the bitmap to which it is being blitted. In this case, excessonoff allows you to specify an on/off value for the excess bitplanes – ie, the bitplanes beyond those altered by the shape. Bit zero of excessonoff will specify an on/off value for the first excess bitplane, bit one an on/off value for the second excess bitplane and so on.

The manner in which the shape is drawn onto the bitmap may be altered by use of the BlitMode command.

BlitMode bltcon0

Allows you to specify just how the Blit command uses the blitter when drawing shapes to bitmaps. By default, BlitMode is set to a 'CookieMode' which simply draws shapes 'as is'. However, this mode may be altered to produce other useful ways of drawing. Here are just some of the possible BLTCON0 parameters:

CookieMode:	Shapes are drawn 'as is'.
EraseMode:	An area the size and shape of the shape will be 'erased' on the destination bitmap.
InvMode:	An area the size and shape of the shape will be inverted on the destination bitmap.
SolidMode:	The shape will be drawn as a solid area of one colour.

Actually, these modes are all just special functions which return a useful value. Advanced programmers may be interested to know that the BLTCON0 parameter is used by the Blit command's blitter routine to determine the blitter MINITERM and CHANNEL USE flags. Bits zero through seven specify the miniterm, and bits eight through eleven specify which of the blitter channels are used. For the curious out there, all the blitter routines in Blitz assume the following blitter channel setup:

BlitterChannel	Used For
A	Pointer to shape's cookie cut
B	Pointer to shape data
C	Pointer to destination
D	Pointer to destination

CookieMode

This 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 cleanly, or 'as is', onto a bitmap.

EraseMode

This function returns a value which may be used by one the commands involved in blitting modes.

Using EraseMode as a blitting mode will cause a blitted shape to erase a section of a bitmap corresponding to the outline of the shape.

InvMode

This function returns a value which may be used by one 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 blitted shape.

SolidMode

The SolidMode function returns a value which may be used by one 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 blitted shape.

Queue queue#,maxitems

Creates a queue object for use with the QBlit and UnQueue commands. What is a queue? Well, queues (in the Blitz sense) are used for the purpose of multi-shape animation. Before going into what a queue is, let's have a quick look at the basics of animation.

Say you want to get a group of objects flying around the screen. To achieve this, you will have to construct a loop similar to the following:

```
Step 1: Start at the first object
Step 2: Erase the object from the display
Step 3: Move the object
Step 4: Draw the object at its new location on the display
Step 5: If there are any more objects to move, move on to the next object and then go
        to Step 2, else...
Step 6: Go to step 1
```

Step 2 is very important, as if it is left out, all the objects will leave trails behind them! However, it is often very cumbersome to have to erase every object you wish to move. This is where queues are of use.

Using queues, you can 'remember' all the objects drawn through a loop, then, at the end of the loop (or at the start of the next loop), erase all the objects 'remembered' from the previous loop. Look at how this works:

```
Step 1: Erase all objects remembered in the queue
Step 2: Start at the first object
Step 3: Move the object
Step 4: Draw the object at its new location, and add it to the end of the queue
Step 5: If there are any objects left to move, go on to the next object, then go to
        step 3; else...
Step 6: Go to step 1
```

This is achieved quite easily using Blitz's queue system. The UnQueue command performs step 1, and the QBlit command performs step 4.

Queues purpose is to initialise the actual queue used to remember objects in. Queue must be told the maximum number of items the queue is capable of remembering, which is specified in the maxitems parameter.

QBlit queue#,shape#,x,y[,excessonoff]

Performs similarly to Blit, and is also used to draw a shape onto the currently used bitmap. Where QBlit differs, however, is in that it also remembers (using a queue) where the shape was drawn, and how big it was. This allows a later UnQueue command to erase the drawn shape.

The optional excessonoff parameter works identically to the excessonoff parameter used by the Blit command.

UnQueue queue#[,bitmap#]

Used to erase all ‘remembered’ items in a queue. Items are placed in a queue by use of the QBlit command.

An optional bitmap# parameter may be supplied to cause items to be erased by way of ‘replacement’ from another bitmap, as opposed to the normal ‘zeroing out’ erasing.

FlushQueue queue#

Will force the specified queue object to be ‘emptied’, causing the next UnQueue command to have no effect.

QBlitMode BLTCON0

Allows you to control how the blitter operates when QBlitting shapes to bitmaps.

Buffer buffer#,memorylen

Used to create a buffer object. Buffers are similar to queues in concept, but operate slightly differently. If you have not yet read the description of the Queue command, it would be a good idea to do so before continuing here.

The buffer related commands are very similar to the queue related commands – Buffer, BBlit, and UnBuffer, and are used in exactly the same way. Where buffers differ from queues, however, is in their ability to preserve background graphics. Whereas an UnQueue command normally trashes any background graphics, UnBuffer will politely restore whatever the BBlits may have overwritten. This is achieved by the BBlit command actually performing two blits.

The first blit transfers the area on the bitmap which the shape is about to cover to a temporary storage area - the second blit actually draws the shape onto the bitmap. When the time comes to UnBuffer all those BBlits, the temporary storage areas will be transferred back to the disrupted bitmap.

The memorylen parameter of this command refers to how much memory, in bytes, should be put aside as temporary storage for the preservation of background graphics. The value of this parameter varies depending upon the size of shapes to be blitted, and the maximum number of shapes to be blitted between UnBuffers. A memorylen of 16384 should be plenty for most situations, but may need to be increased if you start getting 'Buffer Overflow' error messages.

BBlit buffer#,shape#,x,y[,excessonoff]

The BBlit command is used to draw a shape onto the currently used bitmap, and preserve the overwritten area into a previously initialised buffer.

The optional excessonoff parameter works identically to the excessonoff parameter used by the Blit command.

UnBuffer buffer#

Used to 'replace' areas on a bitmap overwritten by a series of BBlit commands. For more information on buffers, please refer to the Buffer command.

FlushBuffer buffer#

Will force the specified buffer object to be 'emptied', causing the next UnBuffer command to have no effect.

BBlitMode bltcon0

Allows you to control how the blitter operates when BBlitting shapes to bitmaps.

Stencil stencil#,bitmap#

Creates a stencil object based on the contents of a previously created bitmap. The stencil will contain information based on all graphics contained in the bitmap, and may be used with the SBlit and ShowStencil commands.

SBlit stencil#,shape#,x,y[,excessonoff]

Works identically to the Blit command, and also updates the specified stencil#. This is an easy way to render 'foreground' graphics to a bitmap.

SBlitMode bltcon0

Used to determine how the SBlit command operates. Please refer to the BlitMode command tor more information on blitting modes.

ShowStencil buffer#,stencil#

Used in connection with BBlits and stencil objects to produce a ‘stencil’ effect. Stencils allow you create the effect of shapes moving ‘between’ background and foreground graphics. Used properly, stencils can add a sense of ‘depth’ or ‘three dimensionality’ to animations.

So what steps are involved in using stencils? To begin with, you need both a bitmap and a stencil object. A stencil object is similar to a bitmap in that it contains various graphics. Stencils differ, however, in that they contain no colour information. They simply determine where graphics are placed on the stencil. The graphics on a stencil usually correspond to the graphics representing ‘foreground’ scenery on a bitmap.

So the first step is to set up a bitmap with both foreground and background scenery on it. Next, a stencil is set up with only the foreground scenery on it. This may be done using either the Stencil or SBlit command. Now, we BBlit our shapes. This will, of course, place all the shapes in front of both the background and the foreground graphics. However, once all shapes have been blitted, executing the ShowStencil command will repair the damage done to the foreground graphics!

Block shape#,x,y

An extremely fast version of the Blit command with some restrictions. Block should only be used with shapes that are 16,32,48,64...pixels wide and that are being blitted to an x position of 0,16,32,48,64...

Note: the height and y destination of the shape are not limited by the Block command. Block is intended tor use with map type displays.

BlitColl(shape#,x,y)

A fast way of collision detection when blitting shapes. BlitColl returns -1 if a collision occurs, 0 if no collision. A collision occurs if any pixel on the current bitmap is non zero where your shape would have been blitted.

ShapesHit is faster but less accurate as it checks only the rectangular area of each shape, where as BlitColl takes into account the shape of the shape and of course can not tell you what shape you have collided with.

ClipBlit clipblit shape#,x,y

Same as the Blit command except ClipBlit will clip the shape to the inside of the used bitmap, all blit commands in Blitz are due to be expanded with this feature.

ClipBlitMode bplcon0

Same as BlitMode except applies to the ClipBlit command. Another oversight now fixed.

BlockScroll x1,y1,width,height,x2,y2[,bitmap#]

Same as the Scroll command except that BlockScroll is much faster but only works with 16 bit aligned areas. This means that x1, y2 and width must all be multiples of 16. Useful for block scrolling routines that render the same blocks to both sides of the display, the programmer can now choose to render just one set and then copy the result to the other side with the BlockScroll command.

R-18: SPRITE HANDLING

Sprites are another way of producing moving objects on the Amiga's display. Sprites are, like shapes, graphical objects. However unlike shapes, sprites are handled by the Amiga's hardware completely separately from bitmaps. This means that sprites do not have to be erased when its time to move them, and that sprites in no way destroy or interfere with bitmap graphics. Also, once a sprite has been displayed, it need not be referenced again until it has to be moved.

In this release of Blitz, sprites are only available in Blitz mode and have either 3 or 15 colours (2 or 4 bitplanes). Each slice may display a maximum of up to 8 sprites. Other conditions may lower this maximum such as the width, depth and resolution of the slice. The Amiga hardware has 8 sprite channels, standard 16 wide 3 colour sprites require a single channel, 15 colour sprites need two and sprites wider than 16 will require extra channels also. 15 color sprites must use an even numbered channel, the subsequent odd channel then becomes unavailable.

Sprites also require a special colour palette set up. Fifteen colour sprites take their RGB values from colour registers 17 through 31. Three colour sprites, however, take on RGB values depending upon the sprite channels being used to display them. The following table shows which palette registers affect which sprite channels:

Sprite Channel	Colour Registers
0,1	17-19
2,3	21-23
4,5	25-27
6,7	29-31

GetaSprite sprite#,shape#

To be able to display a sprite, you must first create a sprite object. This will contain the image information for the sprite. GetaSprite will transfer the graphic data contained in a shape object into a sprite object. This allows you to perform any of the Blitz shape manipulation commands (eg Scale or Rotate) on a shape before creating a sprite from the shape.

Once GetaSprite has been executed, you may not require the shape object any more. In this case, it is best to free up the shape object (using Free Shape) to conserve as much valuable chip memory as possible.

ShowSprite sprite#,x,y,spritechannel

Used to actually display a sprite through a sprite channel. Paremeters x and y specify the position the sprite is to be displayed at. These parameters are ALWAYS given in lo-resolution pixels. Spritechannel is a value 0 through 7 which decides which sprite channel the sprite should be display through.

InFront spritechannel

A feature of sprites is that they may be displayed either ‘in front of’ or ‘behind’ the bitmap graphics they are appearing in. This command allows you to determine which sprites appear in front of bitmaps, and which sprites appear behind.

Spritechannel must be an even number in the range 0 through 8. After executing an InFront command, sprites displayed through sprite channels greater than or equal to spritechannel will appear BEHIND any bitmap graphics. Sprites displayed through channels less than spritechannel will appear IN FRONT OF any bitmap graphics. For example, after executing an InFront 4, any sprites displayed through sprite channels 4,5,6 or 7 will appear behind any bitmap graphics, while any sprites displayed through sprite channels 0,1,2 or 3 will appear in front of any bitmap graphics.

InFront should only be used in non-dualplayfield slices.

InFrontF spritechannel

Used on dualplayfield slices to determine sprite/playfield priority with respect to the foreground playfield. Using combinations of InFrontF and InFrontB (used for the background playfield), it is possible to display sprites at up to 3 different depths - some in front of both playfields, between the playfields, and behind both playfields.

InFrontB spritechannel

Used on dualplayfield slices to determine sprite/playfield priority with respect to the background playfield. Using combinations of InFrontB and InFrontF (used tor the foreground playfield), it is possible to display sprites at up to 3 different depths - some in front of both playfields, some between the playfields, and some behind both playfields.

LoadSprites sprite#[,sprite#],filename$

Lets you load a ‘range’ of sprites from disk into a series of sprite objects. The file specified by filename$ should have been created using the SaveSprites command. The first sprite# parameter specifies the number of the first sprite object to be loaded. Further sprites will be loaded into increasingly higher sprite objects. If a second sprite# parameter is supplied, then only sprites up to and including the second sprite# value will be loaded. If there are not enough sprites in the file to fill this range, any excess sprites will remain untouched.

SaveSprites sprite#,sprite#,filename$

Allows you to create a file containing a range of sprite objects. This file may be later loaded using the LoadSprites command. The range of sprites to be saved is specified by sprite#,sprite#, where the first sprite# refers to the lowest sprite to be saved and the second sprite# the highest.

SpriteMode mode

For use with the capabilities of the new display library SpriteMode is used to define the width of sprites to be used in the program. The mode values 0, 1 and 2 correspond to the widths 16, 32 and 64.

R-19: COLLISION DETECTION

This section deals with various commands involved in detection of object collisions.

SetColl colour,bitplanes[,playfield]

There are 3 different commands involved in controlling sprite/bitmap collision detection, of which SetColl is one (the other 2 being SetCollOdd and SetCollHi). All three determine what colours in a bitmap will cause a collision with sprites. This allows you to design bitmaps with 'safe' and 'unsafe' areas.

SetColl allows you to specify a single colour which, when present in a bitmap, and in contact with a sprite, will cause a collision. The colour parameter refers to the collide-able colour. Bitplanes refers to the number of bitplanes (depth) that bitmap collision are to be tested in.

The optional playfield parameter is only used in a dualplayfield slice. If playfield is 1, then colour refers to a colour in the foreground bitmap. If playfield is 0, then colour refers to a colour in the background bitmap.

DoColl and PColl are the commands used for actually detecting the collisions.

SetCollOdd

Used to control the detection of sprite/bitmap collisions. SetCollOdd will cause ONLY the collisions between sprites and 'odd coloured' bitmap graphics to be reported. Odd coloured bitmap graphics refers to any bitmap graphics rendered in an odd colour number (de: 1,3,5...). This allows you to design bitmap graphics in such a way that even coloured areas are 'safe' (de: they will not report a collision) whereas odd colour areas are 'unsafe' (de: they will report a collision).

DoColl and PColl commands are used to detect the actual sprite/bitmap collisions.

SetCollHi bitplanes

May be used to enable sprite/bitmap collisions between sprites and the 'high half' colour range of a bitmap. For example, if you have a 16 colour bitmap, the high half of the colours would be colours 8 through 15. The bitplanes parameter should be set to the number of bitplanes (depth) of the bitmap with which collisions should be detected.

Please refer to the SetColl command for more information on sprite/bitmap collisions.

DoColl

Used to perform sprite/bitmap collision checking. Once DoColl is executed, the PColl and/or SColl functions may be used to check for sprite/bitmap or sprite/sprite collisions. Before DoColl may be used with PColl, the type of bitmap collisions to be detected must have been specified using one of the SetColl, SetCollOdd or SetCollHi commands.

After executing a DoColl, PColl and SColl will return the same values until the next time DoColl is executed.

PColl(spritechannel)

This function may be used to find out if a particular sprite has collided with any bitmaps. Spritechannel refers to the sprite channel of the sprite you wish to check is being displayed through. If the specified sprite has collided with any bitmap graphics, PColl will return a true (-1) value, otherwise PColl will return false (0).

Before using PColl, a DoColl must previously have been executed.

SColl(spritechannel,spritechannel)

Used to determine whether the 2 sprites currently displayed through the specified sprite channels have collided. If they have, SColl will return true (-1), otherwise SColl will return false (0).DColl must have been executed prior to using Scoll.

ShapesHit(shape#,x,y,shape#,x,y)

This function will calculate whether the rectangular areas occupied by 2 shapes overlap. ShapesHit will automatically take the shape handles into account. If the 2 shapes overlap, ShapesHit will return true (-1),otherwise ShapesHit will return false (0).

ShapeSpriteHit(shape#,x,y,sprite#,x,y)

This function will calculate whether the rectangular area occupied by a shape at one position, and the rectangular area occupied by a sprite at another position are overlapped. If the areas do overlap, ShapeSpriteHit will return true (-1), otherwise ShapeSpriteHit will return false (0). ShapeSpriteHit automatically takes the handles of both the shape and the sprite into account.

SpritesHit(sprite#,x,y,sprite#,x,y)

This function will calculate whether the rectangular areas occupied by 2 sprites overlap. SpritesHit will automatically take the sprite handles into account. If the 2 sprites overlap, SpritesHit will return true (-1), otherwise SpritesHit will return false (0).

Care should be taken with the pronunciation of this command.

RectsHit(x1,y1,width1,height1,x2,y2,width2,height2)

This function may be used to determine whether 2 arbitrary rectangular areas overlap. If the specified rectangular areas overlap, RectsHit will return true (-1), otherwise RectsHit will return false (0).

Care should be taken with the pronunciation of this command.

R-20: PALETTES

Amiga colours are represented as values for the three primary colours red, green and blue. These values are combined as an RGB value. Palettes are Blitz objects that contain a series of RGB values that represent the colours used by the display. Palette information can be loaded from an IFF file or defined using the PalRGB/AGAPalRGB commands. Palettes can be assigned to screens and slices with both the Use Palette and ShowPalette commands.

Many commands are available for manipulating the colours within a palette.

Colour values on slices and screens can also be changed directly without the use of palettes using the RGB and AGARGB commands.

LoadPalette palette#,filename$[,paletteoffset]

Creates and initialises a palette object. Filename$ specifies the name of an ILBM IFF file containing colour information. If the file contains colour cycling information, this will also be loaded into the palette object.

An optional paletteoffset may be specified to allow the colour information to be loaded at a specified point (colour register) in the palette. This is especially useful in the case of sprite colours, as these must begin at colour register sixteen.

LoadPalette does not actually change any display colours. Once a palette is loaded, Use Palette can be used to cause display changes.

ShowPalette palette#

Replaces Use Palette for copying a palette’s colours to the current screen or slice.

Use Palette palette#

Transfers palette information from a palette object to a displayable palette. If executed in Amiga mode, palette information is transferred into the palette of the currently used screen. If executed in Blitz mode, palette information is transferred into the palette of the currently used slice.

NewPaletteMode On|Off

This flag has been added for compatibility with older Blitz programs. By setting NewPaletteMode to On the Use Palette command merely makes the specified palette the current object and does not try to copy the colour information to the current screen or slice.

Free Palette palette#

Erases all information in a palette object. That palette object may no longer be Used or Cycled.

SavePalette palette#,filename$

Creates a standard IFF "CMAP" file using the given palette’s colours.

CyclePalette palette#

Uses the standard color cycling parameters in the palette object to cycle the colours. Unlike the Cycle command which copied the resulting palette to the current screen the CyclePalette command just modifies the palette object and can hence be used with the DisplayBitmap command in the new display library.

FadePalette srcpalette#,destpalette#,brightness.q ;palettelib

Multiplies all colours in a palette by the brightness argument and maces the result in the destpalette.

lnitPalette palette#,numcolours

Simply initialises a palette object to hold numcolours. All colours will be set to black.

DecodePalette palette#,memorylocation[,paletteoffset]

Allows the programmer to unpack included IFF palette information to Blitz palette objects.

PalRGB palette#,colourregister,red,green,blue

Allows you to set an individual colour register within a palette object. Unless an RGB has also been executed, the actual colour change will not come into effect until the next time ShowPalette is executed.

RGB colourregister,red,green,blue

Enables you to set individual colour registers in a palette to an RGB colour value. If executed in Amiga mode, RGB sets colour registers in the currently used screen. If executed in Blitz Mode, RGB sets colour registers in the currently used slice.

Note that RGB does not alter palette objects in any way.

Red(colourregister)

Returns the amount of RGB red in a specified colour register. If executed in Amiga mode, Red returns the amount of red in the specified colour register of the currently used screen. If executed in Blitz mode, Red returns the amount of red in the specified colour register of the currently used slice.

This command will always return a value between 0 and 15.

Green(colourregister)

Returns the amount of RGB green in a specified colour register. If executed in Amiga mode, Green returns the amount of green in the specified colour register of the currently used screen. If executed in Blitz mode, Green returns the amount of green in the specified colour register of the currently used slice.

This command will always return a value between 0 and 15.

Blue(colourregister)

Returns the amount of RGB blue in a specified colour register. If executed in Amiga mode, Blue returns the amount of blue in the specified colour register of the currently used screen. If executed in Blitz mode, Blue returns the amount of blue in the specified colour register of the currently used slice.

This command will always return a value between 0 and 15.

AGARGB colourregister,red,green,blue

This is the AGA equivalent of the RGB command. The red, green and blue parameters must be in the range 0 through 255, while colourregister is limited to the number of colours available on the currently used screen.

AGAPalRGB palette#,colourregister,red,green,blue

This is the AGA equivalent of the PalRGB command. AGAPalRGB allows you to set an individual colour register within a palette object. This command only sets up an entry in a palette object, and will not alter the actual screen palette until a 'ShowPalette' is executed.

AGARed(colourregister)

Returns the red component of the specified colour register within the currently used screen. The returned value will be within the range 0 (being no red) through 255 (being full red).

AGAGreen(colourregister)

Returns the green component of the specified colour register within the currently used screen. The returned value will be within the range 0 (being no green) through 255 (being full green).

AGABlue(colourregister)

Returns the blue component of the specified colour register within the currently used screen. The returned value will be within the range 0 (being no blue) through 255 (being full blue).

SetCycle palette#,cycle,lowcolour,highcolour[,speed]

Used to configure colour cycling information for the Cycle command. Low and high colours specify the range of colours that will cycle. You may have a maximum of 7 different cycles for a single palette. The optional parameter speed specifies how quickly the colours will cycle, a negative value will cycle the colours backwards.

Cycle palette#

Causes the colour cycling information contained in the specified palette to be cycled on the currently used screen. Colour cycling information is created when LoadPalette is executed or with the SetCycle command. StopCycle will halt all colour cycling started with the Cycle command.

FadeIn palette#[,rate[,lowcolour,highcolour]]

Causes the colour palette of the currently used slice to be ‘faded in’ from black up to the RGB values contained in the specified palette#.

The rate# allows to control the speed of the fade, with 0 being the fastest fade. Lowcolour and highcolour allow to control which colour palette registers are affected by the fade.

FadeOut palette#[,rate[,lowcolour,highcolour]]

Causes the colour palette of the currently used slice to be ‘faded out’ from the RGB values contained in the specified palette# down to black.

The rate# allows to control the speed of the fade, with 0 being the fastest fade. Lowcolour and highcolour allow to control which colour palette registers are affected by the fade. For FadeOut to work properly, the RGB values in the currently used slice should be set to the specified palette# prior to using FadeOut.

ASyncFade On|Off

Allows you control over how the FadeIn and FadeOut commands work. Normally, FadeIn and FadeOut will halt program flow, execute the entire fade, and then continue program flow. This is ASyncFade Off mode.

ASyncFade On will cause FadeIn and FadeOut to work differently. Instead of performing the whole fade at once, the programmer must execute the DoFade command to perform the next step of the fade. This allows fading to occur in parallel with program flow.

DoFade

Causes the next step of a fade to be executed. ASyncFade On, and a FadeIn or FadeOut must be executed prior to calling DoFade.

The FadeStatus function may be used to determine whether there are any steps of fading left to perform.

FadeStatus

Used in conjunction with the DoFade command to determine if any steps of fading have yet to be performed. If a fade process has not entirely finished yet (ie: more DoFades are required), then FadeStatus will return true (-1). If not, FadeStatus will return false (0). Please refer to ASyncFade and DoFade for more information.

PaletteRange palette#,startcol,endcol,r0,g0,b0,r1,g1,b1

Creates a spread of colours within a palette. Similar to DPaint's spread function, PaletteRange takes a start and end colour and creates the color tweens between both of them them.

DuplicatePalette srcpalette#,destpalette#

Creates a new palette which exactly matches the srcpalette.

Sound objects are used to store audio information. This information can be taken from an 8SVX IFF file using LoadSound, or defined by hand through a BASIC routine using InitSound and SoundData. Once a sound is created, it may be played through the Amiga's audio hardware.

Blitz supports loading and playing of both SoundTracker and MED module music files.

The Amiga speech synthesiser is also accessible from Blitz. The narrator.device has been upgraded in 2.0 increasing the quality of the speech. With a bit of messing around you can have a lot of fun with the Amiga's 'voice'.

LoadSound sound#,filename$

Creates a sound object for later playback. The sound is taken from an 8SVX IFF file. An error will be generated if the specified file is not in the correct IFF format.

Sound sound#,channelmask[,vol1[,vol2...]]

Causes a previously created sound object to be played through the Amiga's audio hardware. Channelmask specifies which of the Amiga's four audio channels the sound should be played through, and should be in the range 1 through 15.

The following is a list of Channelmask values and their effect:

Mask	Channel0	Channel1	Channel2	Channel3
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

In the above table, any audio channels specified as ‘off’ are not altered by Sound, and any sounds they may have previously been playing will not be affected.

The volx parameters allow individual volume settings for different audio channels. Volume settings must be in the range zero through 64, zero being silence, and 64 being loudest. The first vol parameter specifies the volume for the lowest numbered ‘on’ audio channel, the second vol for the next lowest and so on.

For example, assume you are using the following Sound command:

```
Sound 0,10,32,16
```

The channelmask of ten means the sound will play through audio channels one and three. The first volume of 32 will be applied to channel one, and the second volume of 16 will be applied to channel three. Any vol parameters omitted will be cause a volume setting of 64.

LoopSound sound#,channelmask[,vol1[,vol2...]]

Behaves identically to Sound, only the sound will be played repeatedly. Looping a sound allows for the facility to play the entire sound just once, and begin repeating at a point in the sound other than the beginning. This information is picked up from the 8SVX IFF file, when LoadSound is used to create the sound, or from the offset parameter of InitSound.

Volume channelmask,vol1[,vol2...]

Allows you to dynamically alter the volume of an audio channel. This enables effects such as volume fades. For an explanation of channelmask and vol parameters, please refer to the Sound command.

InitSound sound#,length[,period[,repeat]]

Initialises a sound object in preparation for the creation of custom sound data. This allows simple sound waves such as sine or square waves to be algorithmically created. SoundData should be used to create the actual wave data.

Length refers to the length in bytes, the sound object is required to be. Length MUST be less than 128K, and MUST be even.

Period allows you to specify a default pitch for the sound. A period of 428 will cause the sound to be played at approximately middle ‘C’.

Offset is used in conjunction with LoopSound, and specifies a position in the sound at which repeating should begin. Please refer to LoopSound for more information on repeating sounds.

SoundData sound#,offset,data

Allows you to manually specify the waveform of a sound object. The sound object should normally have been created using InitSound, although altering IFF sounds is perfectly legal. SoundData alters one byte of sound data at the specified Offset. Data refers to the actual byte to place into the sound, and should be in the range -128 to +127.

PeekSound(sound#,offset)

Returns the byte of a sample at specified offset of sound object specified.

DecodeSound sound#,memorylocation

Similar to the other new Decode commands allows the programmer to include sound files within their program's object code.

SetPeriod sound#,period

Allows the programmer to manually adjust the period of the sound object to change its effective pitch.

DiskPlay filename$,channelmask[,vol1[,vol2...]]

Plays an 8SVX IFF sound file straight from disk. This is ideal for situations where you simply want to play a sample without the extra hassle of loading a sound, playing it, and then freeing it. This command will also halt program flow until the sample has finished playing. DiskPlay usually requires much less memory to play a sample than the LoadSound, Sound technique. DiskPlay also allows you to play samples of any length, whereas LoadSound only allows samples up to 128K in length to be loaded.

DiskBuffer bufferlen

Allows you to set the size of the memory buffer used by the DiskPlay command. This buffer is by default set to 1024 bytes, and should not normally have to be set to more than this. Reducing the buffer size by too much may cause loss of sound quality of the DiskPlay command. If you are using DiskPlay to access a very slow device, the buffer size may have to be increased.

Filter On|Off

Filter may be used to turn on or off the Amiga's low pass audio filter.

LoadModule module#,filename$

Loads in from disk a SoundTracker/NoiseTracker music module. This module may be later played back using PlayModule.

Free Module module#

Used to delete a module object. Any memory occupied by the module will also be free'd.

PlayModule module#

Causes a previously loaded SoundTracker/NoiseTracker song module to be played back.

StopModule

Cause any SoundTracker/NoiseTracker modules which may be currently playing to stop.

LoadMedModule medmodule# name

Loads any version 4 channel OctaMED module. Following routines support up to and including version 3 of the Amiganut's MED standard. The number of MedModules loaded in memory at one time is only limited by the MedModules maximum set in the Blitz Options requester. Like any Blitz commands that access files LoadMedModule can only be used in Amiga mode.

StartMedModule medmodule#

Responsible for initialising the module including linking after it is loaded from disk using the LoadMedModule command. It can also be used to restart a module from the beginning.

PlayMed

Responsible for playing the current MedModule, it must be called every 50th of a second either on an interrupt (#5) or after a VWait in a program loop.

StopMed

Cause any med module to stop playing. This not only means that PlayMed will have no affect until the next StartMedModule but silences audio channels so they are not left ringing as is the effect when PlayMed is not called every vertical blank.

JumpMed pattern#

Changes the pattern being played in the current module.

SetMedVolume volume

Changes the overall volume that the MED Library plays the module, all the audio channels are affected. This is most useful for fading out music by slowly decreasing the volume from 64 to 0.

GetMedVolume channel#

Returns the current volume setting of the specified audio channel. This is useful for graphic effects required to sync to certain channels of the music playing.

GetMedNote Channel#

Returns the current note playing from the specified channel. As with GetMedVolume this is useful for producing graphics effects synced to the music the MED Library is playing.

GetMedInstr channel

Returns the current instrument playing through the specified audio channel.

SetMedMask channel Mask

Allows the user to mask out audio channels needed by sound effects stopping the MED Library using them.

DecodeMedModule medmodule#,memorylocation

Replaces the cludgemedmodule, as MED modules are not packed but used raw, DecodeMedModule simply checks to see the memory location passed is in chip mem (if not it copies the data to chip) and points the Blitz MedModule object to that memory.

Speak string$

Will first convert the given string to phonetics and then pass it to narrator.device. Depending on the settings of the narrator device (see SetVoice) the Amiga will "speak" the string you have sent in the familiar Amiga synthetic voice.

SetVoice rate,pitch,expression,sex,volume,frequency

Alters the sound of the Amiga’s speech synthesiser by changing the vocal characteristics listed in the parameters above.

Translate$(string$)

Returns the phonetic equivalent of the string for use with the PhoneticSpeak command.

PhoneticSpeak phonetic$

Similar to the Speak command but should only be passed strings containing legal phonemes such as that produced by the Translate$() function.

VoiceLoc

Returns a pointer to the internal variables in the speech synthesiser that enable the user to access new parameters added to the V37 Narrator Device. Formants as referred to in the descriptions are the major vocal tracts and are separated into the parts of speech that produce the bass, medium and treble sounds.

R-22: SCREENS

The following section covers the Blitz commands that let you open and control Intuition based screen objects.

Screen screen#,mode[,title$]
Screen screen#,x,y,width,height,depth,vmode,title$,dpen,bpen[bmap#]

Will open an Intuition screen. The are two formats of the Screen command, a quick format, and a long format. The quick format of the Screen commands involves 3 parameters – screen#, mode and an optional title$.

Screen# specifies the screen object to create.

Mode specifies how many bitplanes the screen is to have, and should be in the range 1 through 6. Adding 8 to Mode will cause a hi-res screen to be opened, as opposed to the default lo-res screen. A hi-res screen may only have from 1 to 4 bitplanes. Adding 16 to Mode will cause an interlaced screen to be opened. Title$ allows you to add a title to the screen.

The long format of Screen gives you much more control over how the screen is opened.

The vmode parameter refers to the resolution of the screen, add the values together to make up the screen mode you require:

hires	= $8000
ham	= $200
superhires	= $20
interlace	= 4
lores	= 0

ShowScreen screen#

Will cause the specified screen object to be moved to front of the display.

WbToScreen screen#

Assigns the Workbench screen a screen object number. This allows you to perform any of the functions that you would normally do own your own screens on the Workbench screen. Its main usage is to allow you to open windows on the Workbench screen. After execution, the Workbench screen will become the currently used screen.

FindScreen screen#[,title$]

This command will find a screen and give it an object number so it can be referenced in your programs. If title$ is not specified, then the foremost screen is found and given the object number screen#. If the title$ argument is specified, then a screen will be searched for that has this name.

After execution, the found screen will automatically become the currently used screen.

LoadScreen screen#,filename$[,palette#]

Loads an IFF ILBM picture into the screen object specified by screen#. The file that is loaded is specified by filename$. You can also choose to load in the colour palette for the screen, by specifying the optional palette#. This value is the object number of the palette you want the pictures colours to be loaded into. For the colours to be used on your screen, you will have to use the statement.

SaveScreen screen#,filename$

Saves a screen to disk as an IFF ILBM file. The screen you wish to save is specified by the screen#, and the name of the file you to create is specified by filename$.

SMouseX

Returns the horizontal position of the mouse relative to the left edge of the currently used screen.

SMouseY

Returns vertical position of the mouse relative to top of the current screen.

ViewPort(screen#)

This function returns the location of the specified screens viewport. The viewport address can be used with graphics.library commands and the like.

ScreenPens activetext,inactivetext,hilight,shadow,activefill,gadgetfill

Configures the 10 default pens used for system gadgets in Workbench 2. Any screens opened after a ScreenPens statement will use the pens defined. This command will have no affect when used with Workbench 1.3 or earlier.

CloseScreen screen#

Has been added for convenience. Same as Free Screen but a little more intuitive (especially for those that have complained about such matters (yes we care)).

HideScreen screen#

Move the screen to the back of all screens open in the system.

BeepScreen screen#

Flash the specified screen.

MoveScreen screen#,deltax,deltay

Move the specified screen by specified amount. Good for system friendly special effects.

ScreenTags screen#,title$[&taglist]
ScreenTags screen#,title$[[,tag,data]...]

Full access to all the Amiga's new display resolutions is now available in Amiga mode by use of ScreenTags command. Following tags are of most interest to programmers.

#Left	$80000021	#BitMap	$8000002E	#ColorMapEntries	$8000003C
#Top	$80000022	#PubName	$8000002F	#Parent	$8000003D
#Width	$80000023	#PubSig	$80000030	#Draggable	$8000003E
#Height	$80000024	#PubTask	$80000031	#Exclusive	$8000003F
#Depth	$80000025	#DisplayID	$80000032	#SharePens	$80000040
#DetailPen	$80000026	#DClip	$80000033	#Interleaved	$80000042
#BlockPen	$80000027	#Overscan	$80000034	#Colors32	$80000043
#Title	$80000028	#ShowTitle	$80000036	#FrontChild	$80000045
#Colors	$80000029	#Behind	$80000037	#BackChild	$80000046
#ErrorCode	$8000002A	#Quiet	$80000038	#LikeWorkbench	$80000047
#Font	$8000002B	#AutoScroll	$80000039	#Reserved	$80000048
#SysFont	$8000002C	#Pens	$8000003A		
#Type	$8000002D	#FullPalette	$8000003B		

ShowBitMap [bitmap#]

This command is the Amiga-mode version of the Show command. It enables you to change a screen's bitmap allowing double buffered (flicker free) animation to happen on a standard Intuition screen. Unlike Blitz mode it is better to do ShowBitMap then VWait to sync up with the Amiga's display, this will make sure the new bitmap is being displayed before modifying the previous bitmap.

R-23: WINDOWS

Windows are the heart of the user friendly Amiga operating system. Not only are they the graphics device used for both user input and display but are the heart of the messaging system that communicates this information to your program by way of the events system.

Typically a Blitz program will either open or find a screen to use, define a list of gadgets and then open a window on the screen with the gadget list attached. It will then wait for an event such as the user selecting a menu or hitting a gadget and act accordingly.

The program can specify which events they wish to receive by modifying the IDCMP flags for the window. Once an event is received Blitz has a wide range of commands for finding out exactly what the user has gone and done.

Blitz also offers a number of drawing commands that allow the programmer to render graphics to the currently used window.

Window window#,x,y,width,height,flags,title$,dpen,bpen[,gadgetlist#]

Opens an Intuition window on the currently used screen. Window# is a unique object number for the new window. X & y refer to the offset from top left of the screen the window is to appear at. Width and height are the size of the window in pixels.

Flags are the special window flags that a window can have when opened. These flags allow for the inclusion of a sizing gadget, drag bar and many other things. The flags are listed as followed, with their corresponding values. To select more than one of these flags, they must be logically OR’d together using the ‘|’ operator.

For example, to open a window with drag bar and sizing gadget which is active once opened, you would specify a flags parameter of $1|$2|$1000.

Title$ is a BASIC string, either a constant or a variable, that you want to be the title of the window.

Dpen is the colour of the detail pen of the window. This colour is used for window title.

Bpen is the block pen of the window. This pen is used for things like the border around the edge of the window.

The optional gadgetlist# is the number of a gadgetlist object you have may want attached to the window.

After the window has opened, it will become the currently used window. The window library has been extended to handle super bitmap windows. SuperBitMap windows allow the window to have its own bitmap which can actually be larger than the window. The two main benefits of this feature are the window's ability to refresh itself and the ability to scroll around a large area "inside" the bitmap.

To attach a bitmap to a window set the SuperBitMap flag in the flags field and include the bitmap# to be attached.

Window Flag	Value	Description
WINDOWSIZING	$0001	Attaches sizing gadget to bottom right corner of the window and allows it to be sized.
WINDOWDRAG	$0002	Allows the window to be dragged with the mouse by its title bar.
WINDOWDEPTH	$0004	Lets windows be pushed behind or in front of other windows.
WINDOWCLOSE	$0008	Attaches a close gadget to the upper left corner of the window.
SIZEBRIGHT	$0010	With GIMMEZERO & ZEROWINDOWSIZING set, this will leave the right hand margin, the width of the sizing gadget, clear, and drawing in window will not extend over this right margin.
SIZEBBOTTOM	$0020	Same as SIZEBRIGHT except it leaves a margin at the bottom of the window, the width of the sizing gadget.
BACKDROP	$0100	This opens the window behind any other window that is already opened. It cannot have the WINDOWDEPTH flag set also, as the window is intended to stay behind all others.
GIMME00	$0400	This flag keeps the windows border separate from the rest of the windows area. Any drawing on the window, extending to the borders, will not overwrite the border. NOTE: Although convenient, this does take up more memory than usual.
BORDERLESS	$0800	Opens a window without any border on it at all.
ACTIVATE	$1000	Activates the window once opened.

Use Window window#

Causes the specified window object to become the currently used window. Use Window also automatically performs a WindowInput and WindowOutput on the specified window.

Free Window window#

Closes down a window. This window is now gone, and can not be accessed any more by any statements or functions. Once a window is closed, you may want to direct the input and output somewhere new, by calling Use Window on another window, DefaultOutput/DefaultInput, or by some other appropriate means. Window# is the window object number to close.

WindowInput window#

Causes any future executions of the Inkey$, Edit$ or Edit functions to receive their input as keystrokes from the specified window object. WindowInput is automatically executed when either a window is opened, or Use Window is executed. After a window is closed (using Free Window), remember to tell Blitz to get its input from somewhere else useful (for example, using another WindowInput command) before executing another Inkey$, Edit$ or Edit function.

WindowOutput window#

Causes any future executions of either the Print or NPrint statements to send their output as text to the specified window object. WindowOutput is automatically executed when either a window is opened, or Use Window is executed.

After a window is closed (using Free Window), remember to send output somewhere else useful (for example, using another WindowOutput command) before executing another Print or NPrint statement.

DefaultlDCMP idcmpflags

Allows you to set the IDCMP flags used when opening further windows. You can change the flags as often as you like, causing all of your windows to have their own set of IDCMP flags if you wish.

A window's IDCMP flags will affect the types of 'events' reportable by the window. Events are reported to a program by means of either the WaitEvent or Event functions. To select more than one IDCMP Flag when using DefaultIDCMP, combine the separate flags together using the OR operator ('|').

Any windows opened before any DefaultIDCMP command is executed will be opened using an IDCMP flags setting of $2|$4|$8|$20|$40|$100|$200|$400|$40000|$80000.

This should be sufficient for most programs.

If you do use DefaultIDCMP for some reason, it is important to remember to include all flags necessary for the functioning of the program. For example, if you open a window which is to have menus attached to it, you MUST set the $100 (menu selected) IDCMP flag, or else you will have no way of telling when a menu has been selected.

IDCMP	FlagEvent
$2	Reported when a window has its size changed.
$4	Reported when a windows contents have been corrupted. This may mean a windows contents may need to be re-drawn.
$8	Reported when either mouse button has been hit.
$10	Reported when the mouse has been moved.
$20	Reported when a gadget within a window has been pushed ‘down’.
$40	Reported when a gadget within a window has been ‘released’.
$100	Reported when a menu operation within a window has occurred.
$200	Reported when the ‘close’ gadget of a window has been selected.
$400	Reported when a keypress has been detected.
$8000	Reported when a disk is inserted into a disk drive.
$10000	Reported when a disk is removed from a disk drive.
$40000	Reported when a window has been ‘activated’.
$80000	Reported when a window has been ‘de-activated’.

AddIDCMP idcmpflags

Allows you to ‘add in’ IDCMP flags to the IDCMP flags selected by DefaultIDCMP. Please refer to DefaultIDCMP for a thorough discussion of IDCMP flags.

SubIDCMP idcmpflags

Allows you to ‘subtract out’ IDCMP flags from the IDCMP flags selected by DefaultIDCMP. Please refer to DefaultIDCMP for a thorough discussion of IDCMP flags.

WaitEvent

Halt program execution until an Intuition event has been received. This event must be one that satisfies the IDCMP flags of any open windows. If used as a function, WaitEvent returns the IDCMP flag of the event (please refer to DefaultIDCMP for a table of possible IDCMP flags). If used as a statement, you have no way of telling what event occurred.

You may find the window object number that caused the event using the EventWindow function.

In the case of events concerning gadgets or menus, further functions are available to detect which gadget or menu was played with. In the case of mouse button events, the MButtons function may be used to discover exactly which mouse button has been hit.

IMPORTANT NOTE: If you are assigning the result of WaitEvent to a variable, MAKE SURE that the variable is a long type variable. For example:

```
MyEvent.l=WaitEvent
```

Event

Works similarly to WaitEvent in that it returns the IDCMP flag of any outstanding windows events. However, Event will NOT cause program flow to halt. Instead, if no event has occurred, Event will return 0.

EventWindow

Used to determine in which window the most recent window event occurred. Window events are detected by use of either WaitEvent or Event commands. The value returned by this command is the window object number in which the most recent window event occurred.

FlushEvents [idcmpglag]

When window events occur in Blitz, they are automatically 'queued' for you. This means that if your program is tied up processing one window event while others are being created, you wont miss out on anything. Any events which may have occurred between executions of WaitEvent or Event will be stored in a queue for later use. There may be situations where you want to ignore this backlog of events. Use FlushEvents to make it.

Executing FlushEvents with no parameters will completely clear Blitz's internal event queue, leaving you with no outstanding events. Supplying an idcmpglag parameter will only clear events of the specified type from the event queue.

GadgetHit

Returns the identification number of the gadget that caused the most recent 'gadget pushed' or 'gadget released' event.

As gadgets in different windows may possibly possess the same identification numbers, you may also need to use EventWindow to tell exactly which gadget was hit.

MenuHit

Returns the identification number of the menu that caused the last menu event. As with gadgets, you can have different menus for different windows with same identification number. Therefore you may also need to use EventWindow to find which window caused the event. If no menus have yet been selected, Menuhit will return -1.

ItemHit

Returns the identification number of the menu item that caused the last menu event.

SubHit

Returns the identification number of the the menu subitem that caused the last menu event. If no subitem was selected, SubHit will return -1.

MButtons

Returns the codes for the mouse buttons that caused the most recent 'mouse buttons' event. If menus have been turned off using Menus Off, then the right mouse button will also register an event and can be read with Mbuttons.

RawKey

Returns the raw key code of a key that caused most recent 'key press' events.

Qualifier

Returns the qualifier of the last key that caused a 'key press' event to occur. A qualifier is a key which alters the meaning of other keys; for example the 'shift' keys. Here is a table of qualifier values and their equivalent keys:

Key	Left	Right
UnQualified	$8000	$8000
Shift	$8001	$8002
Caps Lock Down	$8004	$8004
Control	$8008	$8008
Alternate	$8010	$8020
Amiga	$8040	$8080

A combination of values may occur, if more that one qualifier key is being held down. The way to filter out the qualifiers that you want is by using the logical AND operator.

WPlot x,y,colour

Plots a pixel in the currently used window at the coordinates x,y in the colour specified by colour.

WBox x1,y1,x2,y2,colour

Draws a solid rectangle in the currently used window. The upper left hand coordinates of the box are specified with the x1 and y1 values, and the bottom right hand corner of the box is specified by the values x2 and y2.

WCircle x,y,radius,colour

Allows to draw a circle in currently used window. You specify the centre of the circle with the coordinates x,y. The radius value specifies the radius of the circle you want to draw. The last value, colour specifies what colour the circle will be drawn in.

WEllipse x,y,xradius,yradius,colour

Draws an ellipse in the currently used window. You specify the centre of the ellipse with the coordinates x,y. Xradius specifies the horizontal radius of the ellipse, yradius the vertical radius. Colour refers to the colour in which to draw the ellipse.

WLine x1,y1,x2,y2[,xn,yn...],colour

Wline allows you to draw a line or a series of lines into the currently used window. The first two sets of coordinates x1,y1,x2,y2, specify the start and end points of the initial line. Any coordinates specified after these initial two, will be the end points of another line going from the last set of end points, to this set. Colour is the colour of the line(s) that are going to be drawn.

WCls [colour]

Clears the currently used window to colour 0, or a colour is specified, then it will be cleared to this colour. If the current window was not opened with the GIMMEZEROZERO flag set, then this statement will clear any border or title bar that the window has. The InnerCls statement should be used to avoid these side effects.

InnerCls [colour]

Clears only the inner portion of the currently used window. It will not clear the titlebar or borders as WCls would do if your window was not opened with the GIMMEZEROZERO flag set. If a colour is specified, then that colour will be used to clear the window.

WScroll x1,y1,x2,y2,deltax,deltay

Cause a rectangular area of the currently used window to be moved or ‘scrolled’. X1 and y1 specify the top left location of the rectangle, x2 and y2 the bottom right. The delta parameters determine how far to move the area. Positive values move the area right/down, while negative values move the area left/up.

Cursor thickness

Set the style of cursor that appears when editing strings or numbers with the Edit$ or Edit functions. If Thickness is less than 0, then a block cursor will be used. If the Thickness is greater then 0, then an underline thickness pixels high will be used.

Editat

After executing an Edit$ or Edit function, Editat may be used to determine the horizontal character position of the cursor at the time the function was exited. Through the use of Editat, EditExit, EditFrom and Edit$, simple full screen editors may be put together.

EditFrom [charpos]

Allows you to control how the Edit$ and Edit functions operate when used within windows. If a charpos parameter is specified, then the next time an edit function is executed, editing will commence at the specified character position (0 being the first character position).

Also, editing may be terminated by the use of the ‘return’ key or also by any non-printable character (‘up arrow’ or ‘Esc’) or a window event. When used in conjunction with Editat and EditExit, this allows you to put together simple full screen editors.

If charpos is omitted, Edit$ and Edit return to normal - editing always beginning at character postition 0, and ‘return’ being the only way to exit.

EditExit

Returns the ASCII value of the character that was used to exit a window based Edit$ or Edit function. You can only exit the edit functions with keypresses other than ‘return’ if EditFrom has been executed prior to the edit call.

WindowFont intuifont#

Sets the font for the currently used window. Any further printing to this window will be in the specified font. Intuifont# specifies a previously initialised intuifont object created using LoadFont.

WColour forecolour[,backcolour]

Sets the foreground and background colour of printed text for the currently used window. Any further text printed on this window will be in these colours.

WJam mode

Sets the text drawing mode of the currently used window. These drawing modes allow you to do inverted, complemented and other types of graphics. The drawing modes can be OR'ed together to create a combination of them.

Mode	Description
0	This draws only the foreground colour and leaves the background transparent. Eg For the letter 0, any empty space (inside and outside the letter) will be transparent.
1	This draws both the foreground and background to the window. Eg with the letter 0 again, the 0 will be drawn, but any clear area (inside and outside) will be drawn in the current background colour.
2	This will exlusive or (XOR) the bits of the graphics. Eg Drawing on the same place with the same graphics will cause the original display to return.
4	This allows the display of inverse video characters. If used in conjunction with Jam2, it behaves like Jam2, but the foreground and background colours are exchanged.

Activate window#

Activate the window specified by window#.

Menus On|Off

May be used to turn ALL menus either on or off. Turning menus off may be useful if you wish to read the right mouse button.

WPointer shape#

Allows you to determine the mouse pointer imagery used in the currently used window. Shape# specifies an initialised shape object the pointer is to take its appearance from, and must be of 2 bitplanes depth (4 colours).

WMove x,y

Move the current window to screen position x,y.

WSize width,height

Alters the width and height of the current window to the values specified.

WMouseX

Returns the horizontal x coordinate of the mouse relative to the left edge of the current window. If the current window was opened without the GIMMEZEROZERO flag set, then the left edge is taken as the left edge of the border around the window, otherwise, if GIMMEZEROZERO was set, then the left edge is the taken from inside the window border.

WMouseY

Returns the vertical y coordinate of the mouse relative to the top of the current window. If the current window was opened without the GIMMEZEROZERO flag set, then the top is taken as the top of the border around the window, otherwise, if GIMMEZEROZERO was set, then the top is taken from inside the window border.

EMouseX

Returns the horizontal position of the mouse pointer at the time the most recent window event occurred. Window events are detected using the WaitEvent or Event commands.

EMouseY

Returns vertical position of the mouse pointer at the time the most recent window event occurred. Window events are detected using the WaitEvent or Event.

WCursX

Returns the horizontal location of the text cursor of the currently used window. The text cursor position may be set using Wlocate.

WCursY

Returns the vertical location of the text cursor of the currently used window. The text cursor position may be set using Wlocate.

WLocate x,y

Used to set the text cursor position within the currently used window. X and y are both specified in pixels as offsets from the top left of the window. Each window has its own text cursor position, therefore changing the text cursor position of one window will not affect any other window's text cursor position.

WindowX
Returns the horizontal pixel location of the top left corner of the currently used window, relative to the screen the window appears in.

WindowY
Returns the vertical pixel location of the top left corner of the currently used window, relative to the screen the window appears in.

WindowWidth
Returns the pixel width of the currently used window.

WindowHeight
Returns the pixel height of the currently used window.

InnerWidth
Returns the pixel width of the area inside the border of currently window.

InnerHeight
Returns the pixel height of the area inside the border of currently window.

WTopOff
Returns the number of pixels between the top of the current window border and the inside of the window.

WLeftOff
Returns the number of pixels between the left edge of the current window border and the inside of the window.

SizeLimits minwidth,minheight,maxwidth,maxheight
Sets the limits that any new windows can be sized to with the sizing gadget. After calling this statement, any new windows will have these limits imposed on them.

RastPort(window#)
Returns the specified window's rastport address. Many commands in the graphics.library and the like require a rastport as a parameter.

PositionSuperBitMap x,y

Used to display a certain area of the bitmap in a super bitmap window.

GetSuperBitMap

After rendering changes to a superbitmap window the bitmap attached can also be updated with the GetSuperBitMap. After rendering changes to a bitmap the superbitmap window can be refreshed with the PutSuperBitMap command. Both commands work with the currently used window.

PutSuperBitMap

See GetSuperBitmap description.

WTitle windowtitle$,screentitle$

Used to alter both the current window’s title bar and its screens title bar. Useful for displaying important stats such as program status etc.

CloseWindow window#

Has been added for convenience. Same as Free Window but a little more intuitive (added for those that have complained about such matters).

WPrintScroll

Scroll the current window upwards if the text cursor is below the bottom of the window and adjust the cursor accordingly. Presently WPrintScroll only works with windows opened with the gimme00 flag set (#gimmezerozero=$400).

WBlit shape#,x,y

Used to blit any shape to the current window. Completely system friendly this command will completely clip the shape to fit inside the visible part of the window Use GIMMEZEROZERO windows for clean clipping when the window has title/sizing gadgets.

BitMaptoWindow bitmap#,window#[srox,srcy,destx,desty,width,height]

Copies a bitmap to a window in an operating system friendly manner (what do you expect). The main use of such a command is for programs which use the raw bitmap commands such as the 2D and Blit libraries for rendering bitmaps quickly but require a windowing environment for the user interface.

EventCode

Returns the actual code of the last event received by your program.

EventQualifier

Returns the contents of the qualifier field. Of use with the new GadTools library and some other low level event handling requirements.

WindowTags window#,flags,title$,[&tagList]
WindowTags window#,flags,title$,[[Tag,Data]...]

Similar to ScreenTags, WindowTags allows the advanced user to open a Blitz window with a list of OS tags as described in the documentation for the OS prior to 2.0.

LoadFont intuifont#,fontname.font$,ysize[,style]

Used to load a font from the fonts: directory. Unlike BlitzFonts any size intuifont can be used. The command WindowFont is used to set text output to a certain intuifont in a particular window. This command has been extended with an optional style parameter. The following constants may be combined:

```
#underl ined = 1
#bold        = 2
#italic      = 4
#extended    = 8    ;wider than normal
#colour      = 64   ;hmm use colour version I suppose
```

R-24: GADGETS

Blitz provides extensive support for the creation and use of Intuition gadgets. This is done through the use of GadgetList objects. Each GadgetList may contain one or more of the many types of available gadgets, and may be attached to a window when that window is opened using the Window command.

Following is a table of gadget flags and gadget types which they are relevant to:

Bit#	Value	Meaning	Text	String	Prop	Shape
0	$1	Toggle on/off	yes	no	no	yes
1	$2	Relative to right side of window	yes	yes	yes	yes
2	$4	Relative to bottom of window	yes	yes	yes	yes
3	$8	Size relative to width of window	no	no	yes	no
4	$10	Size relative to height of window	no	no	yes	no
5	$20	Box select	yes	yes	yes	yes
6	$40	Prop gadget has horizontal movement	no	no	yes	no
7	$80	Prop gadget has vertical movement	no	no	yes	no
8	$100	No border around prop gadget	no	no	yes	no
9	$200	Mutually exclusive	yes	yes	no	no
10	$400	Attach to window’s right border	yes	yes	yes	yes
11	$800	Attach to window’s left border	yes	yes	yes	yes
12	$1000	Attach to window’s top border	yes	yes	yes	yes
13	$2000	Attach to window’s bottom border	yes	yes	yes	yes
14	$4000	Use GimmeZeroZero border	yes	yes	yes	yes

Note: If relative right is set, the gadget’s x should be negative, as should it’s y if relative to bottom is set. When relative width or height flags are set, negative width and/or height parameters should be specified as Intuition calculates actual width as windowwidth+gadgetwidth as it does height when relative size flags are set. Mutually exclusive radio button type gadgets DO NOT require Kickstart 2.0 to operate. See ButtonGroup for more information.

The attach flags are for attaching the gadget to one of the windows borders, the GZZGADGET flag is for attaching the gadget to the "outer" rastport/layer of a gimme zero zero window.

Here is an example of setting up some radio button style text gadgets:

```
TextGadget 0,16,16,512,1,"OPTION 1"
Toggle 0,1,on
TextGadget 0,16,32,512,2,"OPTION 2"
TextGadget 0,16,48,512,3,"OPTION 3"
```

Text Gadgets may now be used to create ‘cycling’ gadgets. Again, these gadgets DO NOT require Kickstart 2.0 to work. A text gadget could contain the ‘|’ character in the gadget's text as Blitz will recognise this as a ‘cycling’ gadget. Use the ‘|’ character to separate the options, like this:

```
TextGadget 0|6|6,0|,"HELLO|GOODBYE|SEEYA|"
```

Now, each time this gadget is clicked on, the gadgets text will cycle through ‘HELLO’, ‘GOODBYE’ and ‘SEEYA’. Note that each option is spaced out to be of equal length. This feature should not be used with a GadgetJam mode of 0.

TextGadget gadgetlist#,x,y,flags,id,text$

This command adds a text gadget to a gadgetlist. A text gadget is the simplest type of gadget consisting of a sequence of characters optionally surrounded by a border.

Flags should be selected from the table at the start of the chapter. Boolean gadgets are the simplest type of gadget available. Boolean gadgets are ‘off’ until the program user clicks on them with the mouse, which turns them ‘on’. When the mouse button is released, these gadgets revert back to their ‘off’ state. Boolean gadgets are most often used for ‘Ok’ or ‘Cancel’ type gadgets.

Toggle gadgets differ in that each time they are clicked on they change their state between ‘on’ and ‘off’. For example, clicking on a toggle gadget which is ‘on’ will cause the gadget to be turned ‘off’, and vice versa.

X and y specify where in the window the gadget is to appear. Depending upon the flags setting, gadgets may be positioned relative to any of the 4 window edges. If a gadget is to be positioned relative to either the right or bottom edge of a window, the appropriate x or y parameter should be negative.

The id parameter is an identification value to be attached to this gadget. All gadgets in a gadgetlist should have unique id numbers allowing you to detect which gadget has been selected. An id may be any positive, non-zero number.

Text$ is the actual text you want the gadget to contain.

ButtonGroup group

Allows you to determine which ‘group’ a number of button type gadgets belong to. Following the execution of ButtonGroup, any button gadgets created will be identified as belonging to the spiecified group. The upshot of all this is that button gadgets are only mutually exclusive to other button gadgets within the same group. Group must be a positive number greater than 0. Any button gadgets created before a ButtonGroup command is issued will belong to group 1.

SetGadgetStatus gadgetlist#,id,value

Used to set a cycling text gadget to a particular value, once set, ReDraw should be used to refresh the gadget to reflect its new value.

GadgetPens forecolour[,backcolour]

Determines the text colours used when text gadgets are created using the TextGadget command. The default values used for gadget colours are a foreground colour of 1, and a background colour of 0.

GadgetJam mode

Allows you to determine the text rendering method used when gadgets are created using the TextGadget command. Please refer to the WJam command in the windows chapter for a full description of jam modes available.

SelectMode mode

Used to pre-define how gadget rendering will show a gadget selection. Mode can be 0 for inverse or 1 for box. Use prior to creation of gadgets.

ShapeGadget gadgetlist#,x,y,flags,id,shape#[,shape#]

Allows to create gadgets with graphic imagery. Shape# refers to a shape object containing the graphics you wish the gadget to contain. This command has been extended to allow an alternative image to be displayed when the gadget is selected.

All other parameters are identical to those in TextGadget.

StringGadget gadgetlist#,x,y,flags,id,maxlen,width

Allows to create an Intuition style ‘text entry’ gadget. When clicked on, a string gadget brings up a text cursor, and is ready to accept text entry trom the keyboard.

X and y specifies the gadgets position, relative to the top left of the window it is to appear in. See the beginning of the chapter for the relevant flags for a string gadget.

The id is an identification value to be attached to this gadget. All gadgets in a gadgetlist should have unique id numbers allowing you to detect which gadgets has been selected. It may be any positive, non-zero number.

Maxlen refers to the maximum number of characters which may be entered.

Width refers to how wide (in pixels) the gadget should be. A string gadget may have a width less than the maximum number of characters it may contain, as characters will be scrolled through the gadget when necessary.

You may read the current contents of a string gadget using the StringText function.

StringText$(gadgetlist#,id)

Allows you to determine the current contents of a string gadget. It will return a string of characters representing the string gadgets contents.

ActivateString window#,id

May be used to 'automatically' activate a string gadget. This is identical to the program user having clicked in the string gadget themselves as the string gadget's cursor will appear, and further keystrokes will be sent to the string gadget.

It is often nice of a program to activate important string gadgets as it saves the user the hassle of having to reach the mouse before the keyboard.

ResetString gadgetlist#,id

Allows you to 'reset' a string gadget. This will cause the string gadget's cursor position to be set to the left-most position.

ClearString gadgetlist#,id

May be used to clear or erase the text in the specified string gadget. The cursor position will also be moved to the left-most position in the string gadget. If a string gadget is cleared while it is displayed in a window, the text will not be erased from the actual display. To do this, ReDraw must be executed.

SetString gadgetlist#,id,string$

May be used to initialise the contents of a string gadget created with the StringGadget command. If the string gadget specified by gadgetlist# and id is already displayed, you will also need to execute ReDraw to display the change.

PropGadget gadgetlist#,x,y,flags,id,width,height

Used to create a 'proportional gadget'. Proportional gadgets present a program user with a 'slider bar', allowing them to adjust the slider to achieve a desired effect. For example, proportional gadgets are commonly used for RGB sliders seen in many paint packages.

Proportional gadgets have 2 main qualities - a 'pot' (short for potentiometer) setting, and a body setting.

The pot setting refers to the current position of the slider bar and is in the range 0 through 1. For example, a proportional gadget which has been moved to half way would have a pot setting of '0.5'.

The body setting refers to the size of the units the proportional gadget represents, and is again in the range 0 through 1. Again, taking the RGB colour sliders as an example, each slider is intended to show a particular value in the range 0 through 15 - giving a unit size, or body setting, of 1/16 or '.0625'.

Put simply, the pot setting describes 'where' the slider bar is, while the body setting describes 'how big' it is. Proportional gadgets may be represented as either horizontal slider bars, vertical slider bars, or a combination of both. See the beginning of the chapter for relevant flags settings for prop gadgets.

X and y refer to the gadget position, relative to top left of the window it is opened in.

Width and height refer to the size of the area the slider should be allowed to move in.

The id is a unique, non zero value which allows to identify when the gadget is manipulated. Proportional gadgets may be altered using the SetVProp and SetHProp commands and be read using the VPropPot, VPropBody, HPropPot and HPropBody functions.

SetHProp gadgetlist#,id,pot,body

Used to alter the horizontal slider qualities of a proportional gadget.

Both pot and body should be in the range 0 through 1.

If executed while the specified gadget is already displayed, the ReDraw command will be necessary to display the changes. For a full discussion on proportional gadgets, please refer to the PropGadget command.

SetVProp gadgetlist#,id,pot,body

Used to alter the vertical slider qualities of a proportional gadget. Both pot and body should be in the range 0 through 1.

If executed while the specified gadget is already displayed, ReDraw command will be necessary to display the changes.

HPropPot(gadgetlist#,id)

Returns the current pot setting of a proportional gadget. A number will be returned from 0 up to, but not including, 1, reflecting the gadgets current setting.

HPropBody(gadgetlist#,id)

Returns the current body setting of a proportional gadget. A number will be returnedf from 0 up to, but not including, 1, reflecting the gadgets current setting.

VPropPot(gadgetlist#,id)

Returns the current pot setting of a proportional gadget. A number will be returned from 0 up to, but not including, 1, reflecting the gadgets current setting.

VPropBody(gadgetlist#,id)

Allows you to determine the current 'body' setting of a proportional gadget. A number will be returned from 0 up to, but not including, 1, reflecting the gadgets current setting.

Redraw window#,id

Will redisplay the specified gadget in the specified window. This command is mainly of use when a proportional gadget has been altered using SetHProp or SetVProp and needs to be redrawn, or when a string gadget has been cleared using ClearString, and, likewise, needs to be redrawn.

Borders [On|Off]
Borders [width,height]

This serves two purposes. First, Borders may be used to turn on or off the automatic creation of borders around text and string gadgets. Borders are created when either a TextGadget or StringGadget command is executed. If you wish to disable this, Borders Off should be executed before the appropriate TextGadget or StringGadget command.

Borders may also be used to specify the spacing between a gadget and its border, width referring to the left/right spacing, and height to the above/below spacing.

BorderPens highlightcolour,shadowcolour

Allows you to control the colours used when gadget borders are created. Gadget borders may be created by TextGadget, StringGadget and GadgetBorder. Highlightcolour refers to the colour of the top and left edges of the border, while shadowcolour refers to the right and bottom edges. The default value for highlightcolour is 1 and the default value for shadowcolour is 2.

Gadget Border x,y,width,height

May be used to draw a rectangular border into the currently used window. Proportional gadgets and shape gadgets do not automatically have borders created tor them. This command may be used once a window is opened to render borders around these gadgets.

X, y, width and height refer to the position of the gadget a border is required around. This will automatically insert spaces between the gadget and the border and may be used to alter the amount of spacing. Of course, GadgetBorder may be used to draw a border around any arbitary area, regardless of whether or not that area contains a gadget.

GadgetStatus(gadgetlist#,id)

May be used to determine the status of the specified gadget. In the case of toggle type gadgets, GadgetStatus will return true (-1) if the gadget is currently on, or false (0) if the gadget is currently off. In the case of a cycling text gadget, GadgetStatus will return a value of 1 or greater representing the currently displayed text within the gadget.

ButtonId(gadgetlist#,buttongroup)

Used to determine which gadget within a group of button type gadgets is currently selected. The value returned will be the ID number of the button gadget currently selected.

Enable gadgetlist#,id
Disable gadgetlist#,id

A gadget when disabled is covered by a ‘mesh’ and can not be accessed by the user. Commands Enable & Disable allow the programmer to access this feature of Intuition.

Toggle gadgetlist#,id[,On|Off]

This command in the gadget library has been extended so it will actually toggle a gadgets status if the On|Off parameter is missing.

R-25: MENUS

Blitz supports many commands for the creation and use of Intuition menus which are created through the use of menulist objects. Each menulist contains an entire set of menu titles, menu items and possibly sub menu items and are attached to windows through the SetMenu command. Each window may use a separate menulist, allowing you to attach relevant menus to different windows.

MenuTitle menulist#,menu,title$

Used to add a menu title to a menulist. Menu titles appear when the right mouse button is held down, and usually have menuitems attached to them.

Menu specifies which menu the title should be used for. Higher numbered menus appear further to the right along the menu bar with 0 being the left-most menu. Menu titles should be added in left to right order with menu 0 being the first created, then 1 and so on...

Title$ is the actual text you want to appear when the right mouse button is pressed.

MenuItem menulist#,flags,menu,item,itemtext$[,shortcut$]

Used to create a text menu item. Menu items appear vertically below menu titles when mouse is moved over a menu title with the right mouse button held down.

Flags affects operation of menu item. A value of 0 creates a stand 'select' menu item.

A value of 1 creates a 'toggle' menu item. Toggle menu items are used for 'on/off' type options. When a toggle menu item is selected, it will change state between on and off. An 'on' toggle item is identified by a 'tick' or check mark.

A value of 2 creates a special type of toggle menu item. Any menu items which appear under the same menu with a flags setting of 2 are said to be mutually exclusive. This means that only 1 of them may begin the 'on' state at one time. If a menu item of this nature is toggled into the 'on' state, any other mutually exclusive menu items which may have previously been 'on' will be automatically turned 'off'.

Flags values of 3 and 4 correspond to values 1 and 2, only the item will initially appear in the 'on' state.

Menu specifies the menu title under which the menu item should appear.

Item specifies the menu item number this menu item should be referenced as. Higher numbered items appear further down a menu item list, with 0 being topmost item. Menu items should be added in 'top down' order, with item 0 being the first item created.

Itemtext$ is the actual text for the menu item.

An optional shortcut$ string allows you to select a one character ‘keyboard shortcut’ for the menu item.

ShapeItem menulist#,flags,menu,item,shape#

Used to create a graphical menu item.

Shape# refers to a previously initialised shape object to be used as the menu item’s graphics. All other parameters are identical to those for menuitem.

Subitem menulist#,flags,menu,item,subitem,subitem text$[,shortcut$]

All menu items may have an optional list of sub menu items attached to them.

To attach a sub menu item to a menu item, you use the this command.

Item specifies the menu item to attach the sub item to.

Subitem refers to the number of the sub menu items to attach. Higher numbered sub items appear further down a sub item list with 0 being the topmost sub item. Sub items should be added in ‘top down’ order, with sub item 0 being created first.

Subitemtext$ specifies the actual text for the sub item. As with menu items, sub items may have an optional keyboard shortcut, specified using shortcut$ parameter.

All other parameters are identical to the MenuItem command.

ShapeSub menulist#,flags,menu,item,subitem,shape#

Allows you to create a graphic sub menu item. Shape# specifies a previously created shape object to be used as the sub item’s graphics.

All other parameters are identical to those in SubItem.

SetMenu menulist#

Used to attach a menulist to the currently used window. Each window may have only one menulist attached to it.

MenuGap xgap,ygap

Executing MenuGap before creating any menu titles, items or sub items, allows you to control the layout of the menu.

Xgap refers to an amount, specified in pixels, to be inserted to the left and right of all menu items and sub menu items. Ygap refers to an amount, again in pixels, to be inserted above and below all menu items and sub menu items.

SubItemOff xoffset,yoffset

Allows you to control the relative position of the top of a list of sub menu items, in relation to their associated menu item. Whenever a menu item is created which is to have sub menu items, it's a good idea to append the name of the menu item with the '>>' character. This may be done using Chr$(187). This gives the user a visual indication that more options are available. To position the sub menu items correctly so that they appear after the '>>' character, SubItemOff should be used.

MenuState menulist#[menu[,item[,subitem]]],On|Off

Allows you to turn menus, or sections of menus, on or off. MenuState with just menulist# parameter it's used to turn an entire menu list on or off. MenuState with menulist# and Menu parameters may be used to turn a menu on or off. Similarly, menu items and sub items may be turned on or off by specifying the appropriate parameters.

MenuColour colour

Allows to determine what colour any menu item or sub item text is rendered in. MenuColour should be executed before appropriate menu item commands.

MenuChecked(menulist#,menu,item[,subitem]

Allows you to tell whether or not a 'toggle' type menu item or menu sub item is currently 'checked' or 'on'. If the specified menu item or sub item is in fact checked, it will return 'true' (-1). If not, it will return 'false' (0).

R-26: GADTOOLS

GadTools is a new system of Gadgets added to the Amiga's OS in version 2.0. They are improved in both looks and performance over the older standard gadgets. In order for certain GadTools gadgets to tunction correctly the first thing to make sure is that the window has the correct IDCMP flags set:

```
#MOUSEMOVE=$10 ;needed when user drags a slider
#INTUITICKS=$400000 ;needed
```

When the user holds down an arrow AddIDCMP#MOWSEMOVE+#INTUITICKS.

To add GadTools gadgets to the window simply create a list from the commands listed below and use the AttachGTList command to add them to the window.

For most GTGadgets your program should only act on a #GadgetUp message. The GadgetHit function will return the id of the gadget the user has just hit and the EventCode function will contain its new value.

Use GTGetString and GTGetInteger functions to read the contents of the GadTools string gadgets after a #GadgetUp message.

GadgetFlag	Value	Comment
#_LEFT	1	Position of text label
#_RIGHT	2	
#_ABOVE	4	
#_BELOW	8	
#_IN	$10	
#_Highlight	$20	Gadget is highlighted initially
#_Disable	$40	Gadget is disabled initially
#_Immediate	$80	Report GadgetDown flag
# BoolValue	$100	Gadget is on initially
#_Scaled	$200	Scale arrowsize on scroller gadget
# Vertical	$400	Make GTPropGadget vertical

GTButton gtlist#,id,x,y,w,h,text$,flags

Same as Blitz's TextGadget but with the added flexibility of placing the label text$ above, below, to the left or right of the button (see flags).

GTCheckBox gtlist#,id,x,y,w,h,text$,flags

A box with a check mark that toggles on and off, best used for options that are either enabled or disabled.

GTCycle gtlist#,id,x,y,w,h,text$,flags,options$

Used for offering the user a range of options, the options string should be a list of options separated by the | character. For example, "HIRES|LORES|SUPER HIRES "

GTInteger gtlist#,id,x,y,w,h,text$,flags,default

A string gadget that allows only numbers to be entered by the user. See GTSetInteger and GTGetInteger for information about accessing the contents of a GTInteger gadget.

GTListView gtlist#,id,x,y,w,h,text$,flags,list

A ListView gadget enables user to scroll through a list of options. These options must be contained in a string field of a Blitz linked list. Currently this string field must be the second field, the first being a word type. See the GTChangeList command for more details.

GTMX gtlist#,id,x,y,w,h,text$,flags,options$

This is an exclusive selection gadget, the options$ is the same as GTCycle in format, GadTools then displays all the options in a vertical list each with a hi-light beside them.

GTNumber gtlist#,id,x,y,w,h,text$,flags,value

This is a readonly gadget (the user cannot interact with it) used to display numbers. See GTSetInteger to update the contents of this read only 'display' gadget.

GTPalette gtlist#,id,x,y,w,h,text$,flags,depth

Creates a number of coloured boxes relating to a colour palette.

GTScroller gtlist#,id,x,y,w,h,text$,flags,visible,total

A prop type gadget for the user to control an amount or level, is accompanied by a set of arrow gadgets.

GTSlider gtlist#,id,x,y,w,h,text$,flags,min,max

Same as Scroller but for controlling the position of the display inside a larger view.

GTString gtlist#,id,x,y,w,h,text$,flags,maxchars

A standard string type gadget. See GTSetString and GTGetString for accessing the contents of a GTString gadget.

GTText gtlist#,id,x,y,w,h,text$,flags,display$

A read only gadget (see GTNumber) for displaying text messages. See GTSetString for updating the contents of this read only 'display' gadget.

GTShape gtlist#,id,x,y,flags,shape#[,shape#]

Similar to the Blitz ShapeGadget allowing IFF graphics that are loaded into Blitz shape objects to be used as gadgets in a window.

AttachGTList gtlist#,window#

The AttachGTList command is used to attach a set of GadTools gadgets to a window after it has been opened.

GTTags tag,value,[tag,value...]

Can be used prior to initialisation of any of the 12 GadTools gadgets to preset any relevant tag fields. The following are some useful tags that can be used with GTTags:

Tag	Value	Comment	
#GTCB Checked	$80080004	State of checkbox	
#GTLV_Top	$80080005	Top visible item in listview	
#GTLV_ReadOnly	$80080007	Set TRUE if lisiview is ReadOnly	
#GTMX_Active	$8008000A	Active one in mx gadget	
#GTTX_Text	$8008000B	Text to display	
#GTNM_Number	$8008000C	Number to display	
#GTCY_Active	$8008000F	The active one in the cycle gad	
#GTPA_Color	$80080011	Palette color	
#GTPA ColorOffset	$80080012	First color to use in palette	
#GTSC_Top	$80080015	Top visible in scroller	
#GTSC_Total	$80080016	Total in scroller area	
#GTSC_Visible	$80080017	Number visible in scroller	
#GTSL_Level	$80080028	Slider level	
#GTSL_MaxLevelLen	$80080029	Max length of printed level	
#GTSL_LevelFormat	$8008002A	Format string for level	*
#GTSL_LevelPlace	$8008002B	Where level should be placed	*
#GTLV_Selected	$80080036	Set ordinal number of selected	
#GTMX_Spacing	$8008003D	Added to font height	*

All of the above except for those marked * can be set after initialisation of the gadget using the GTSetAttrs command.

The following is an example of creating a slider gadget with a numeric display:

```
f$="%21d "
GTTags #GTSLLevelFormat,&f$,#GTSLMaxLevelLen,4GTSlider
2,10,320,120,200,20,"GTSLIDER",2,0,10
```

GTGadPtr(gtlist#,id)

Returns the actual location of the specified GadTools gadget in memory.

GTBevelBox gtlist#,x,y,w,h,flags

This is the GadTools library equivalent of the Borders command and can be used to render frames and boxes in the currently used window.

GTChangeList gtlist#,id[,list()]

Must be used whenever a list attached to a GTListView needs to be modified. Call GTChangeList without the list() parameter to free the list, modify it then reattach it with another call to GTChangeList this time using the list() parameter.

GTSetAttrs gtlist#,id[,tag,value...]

Can be used to modify the status of certain GadTools gadgets with the relevant tags. See GTTags for more information.

GTSetString gtlist#,id,string$

Used with both GTString and GTText gadgets, GTSetString will not only update the contents of the gadget but redraw it also.

GTSetInteger gtlist#,id,value

Used with both GTInteger and GTNumber gadgets, GTSetInteger will not only update the contents of the gadget but redraw it also.

GTGetString gtlist#,id

Used to read the contents from a GTString gadget.

GTGetInteger gtlist#,id

Used to read the contents from a GTlnteger gadget.

GTGetAttrs(gtlist#,id,tag)

A 3.0 specific command. See Commodore documentation for more information.

GTEnable gtlist#,id

Allows GTGadgets to be enabled and disabled.

GTDisable gtlist#,id

Allows GTGadgets to be enabled and disabled.

GTToggle gtlist#,id[,On|Off]

Allows the programmer to set boolean gadgets such as GTButton and GTCheckbox to a desired state.

GTStatus(gtlist#,id)

Returns the status of the gadtools toggle gadgets, a value of 1 means the the gadget is selected, 0 deselected.

R-27: ASL LIBRARY

The ASL library features several friendly requesters that programs can use on machines equipped with Workbench 2.0 and above.

ASLFileRequest$(title$,pathname$,filename$[,pattern$][,x,y,w,h])

The ASL file requester is nice. Except for the highlight bar being invisible on directories you get to use keyboard for everything. Stick in a pattern$ to hide certain files and of course you get whatever size you want. I made it call the Blitz file requester if the program is running under 1.3 (isn't that nice!). There's a fix that patches the ReqTools file requester but that doesn't have the date field.

ASLPathRequest$(title$,pathname$[,x,y,w,h])

Same as ASLFileRequest$ except will just prompt the user for a path name (directory) rather than an actual file.

ASLFontRequest(enableflags)

The flags parameter enables the user to modify the following options:

#pen	=	1
#bckgrnd	=	2
#style	=	4
#drawmode	=	8
#fixsize	=	16

It doesn't seem to handle colour fonts, no keyboard shortcuts so perhaps patching ReqTools is an option for this one. The following code illustrates how a .fontinfo structure is created by a call to ASLFontRequest.

ASLScreenRequest(enableflags)

Those who are just getting to grips with 2.0 and above will find this command makes your programs look really good however, I haven't got time to explain the difficulties of developing programs that work in all screen resolutions.

```
NEWTYPE .fontinfo
  name.s
  ysize.w
  style.b:flags.b
  pen1.b:pen2:drawmode:pad
End NEWTYPE

FindScreen 0

*f.fontinfo=ASLFontRequest(15)
If *f
  NPrint  *f\name
  NPrint  *f\ysize
  NPrint  *f\penl
  NPrint  *f\pen2
  NPrint  *f\drawmode
Else
  NPrint "cancelled"
Endif

MouseWait
```

R-28: AREXX

ARexx allows communication between different Amiga applications allowing for some extensive and powerful control over applications by the programmer.

CreateMsgPort (“Name”)

This is a general function and not specific to ARexx.

It opens an Intuition PUBLIC message port of the name supplied as the only argument. If all is well the address of the port created will be returned to you as a LONGWORD so the variable that you assign it to should be of type long. If you do not supply a name then a private MsgPort will be opened for you.

```
Port.l=CreateMsgPort(“PortName”)
```

It is important that you check you actually succeeded in opening a port in your program. The following code or something similar will suffice.

```
Port.l=CreateMsgPort(“Name”)
If Port=0 THEN Error_Routine{ }
```

The name you give your port will be the name that Arexx looks for as the HOST address, (and is case sensitive) so take this into consideration when you open your port. NOTE IT MUST BE A UNIQUE NAME AND SHOULD NOT INCLUDE SPACES. DeleteMsgPort() is used to remove the port later but this is not entirely necessary as Blitz will clean up for you on exit if need be.

DeleteMsgPort(Port)

Deletes a messageport previously allocated with CreateMsgPort(). The only argument taken by DeleteMsgPort is the address returned by CreateMsgPort(). If the port was a public port then it will be removed from the public port list.

```
Port.l=CreateMsgPort(“Name”)
If Port=0 Then End DeleteMsgPort Port
```

Error checking is not critical as if this fails we have SERIOUS PROBLEMS.

YOU MUST WAIT FOR ALL MESSAGES FROM AREXX TO BE RECEIVED BEFORE YOU DELETE THE MSGPORT. IF YOU NEGLECT TO DELETE A MSGPORT BLITZ2 WILL DO IT FOR YOU AUTOMATICALLY ON PROGRAM EXIT.

CreateRexxMsg(replyport,“exten”,“HOST”)

Allocates a special Message structure used to communicate with Arexx. If all is successful it returns the LONGWORD address of this rexxmsg structure.

The arguments are ReplyPort which is the long address returned by CreateMsgPort(). This is the port that ARexx will reply to after it has finished with the message.

EXTEN which is the exten name used by any ARexx script you are wishing to run. ie if you are attempting to run the ARexx script test.rexx you would use an EXTEN of “rexx” HOST is the name string of the HOST port. Your program is usually the HOST and so this equates to the name you gave your port in CreateMsgPort(). REMEMBER IT IS CASE SENSITIVE.

As we are allocating resources error checking is important and can be achieved with the following code:

```
msg.l=CreateRexxMsg(Port,”rexx”,”HostName”) IF msg=0 THEN Error_Routine{}
```

DeleteRexxMsg rexxmsg

Simply deletes a RexxMsg Structure previously allocated by CreateRexxMsg(). It takes a single argument which is the long address of a RexxMsg structure such as a value returned by CreateRexxMsg().

```
msg.l=CreateRexxMsg(Port,”rexx”,”HostName”) IF msg=0 THEN ErrorRoutine{}
DeleteRexxMsg msg
```

Again if you neglect to delete the RexxMsg structure Blitz will do this for you on exit of the program.

ClearRexxMsg Arexxmsg

Used to delete and clear an argstring from one or more of the argument slots in a RexxMsg structure. This is most useful for the more advanced programmer wishing to take advantage of the Arexx #RXFUNC abilities. The arguments are a LONGWORD address of a RexxMsg structure. ClearRexxMsg will always work from slot number 1 forward to 16.

FillRexxMsg(rexxmsg,&fillstruct)

Allows you to fill all 16 ARGSlots if necessary with either argstrings or numerical values depending on your requirement. FillRexxMsg will only be used by those programmers wishing to do more advanced things with ARexx, including adding libraries to the ARexx library list, adding hosts, value tokens etc. It is also needed to access ARexx using the #RXFUNC flag. The arguments are a LONG Pointer to a rexxmsg. The LONG address of a FillStruct NEWTYPE structure. This structure is defined in the Arexx.res and has the following form:

```
NEWTYPE .FillStruct
   Flags.w      ;Flag block
   Args0.l      ;argument block (ARG0-ARG15)
   Args1.l      ;argument block (ARG0-ARG15)
   Args2.l      ;argument block (ARG0-ARG15)
   Args3.l      ;argument block (ARG0-ARG15)
   Args4.l      ;argument block (ARG0-ARG15)
   Args5.l      ;argument block (ARG0-ARG15)
   Args6.l      ;argument block (ARG0-ARG15)
   Args7.l      ;argument block (ARG0-ARG15)
   Args8.l      ;argument block (ARG0-ARG15)
   Args9.l      ;argument block (ARG0-ARG15)
   Args10.l     ;argument block (ARG0-ARG15)
   Args11.l     ;argument block (ARG0-ARG15)
   Args12.l     ;argument block (ARG0-ARG15)
   Args13.l     ;argument block (ARG0-ARG15)
   Args14.l     ;argument block (ARG0-ARG15)
   Args15.l     ;argument block (ARG0-ARG15)
   EndMark.l    ;End of the FillStruct
End NEWTYPE
```

The Args?.l are the 16 slots that can possibly be filled ready for converting into the RexxMsg structure. The Flags.w is a WORD value representing the type of LONG word you are supplying for each ARGSLOT (Arg?.l).

Each bit in the flags WORD is representative of a single Args?.l, where a set bit represents a numerical value to be passed and a clear bit represents a string argument to be converted into a argstring before installing in the RexxMsg. The flags value is easiest to supply as a binary number to make the bits visible and would look like this.

```
%0000000000000000         ;represents that all Arguments are Strings.
%0110000000000000         ;represent second&third as being integers.
```

FillRexxMsg expects to find the address of any strings in the Args?.l slots so it is important to remember when filling a fillstruct that you must pass the string address and not the name of the string. This is accomplished using the ‘&’ address of operand. So to use FillRexxMsg we must do the following things in our program:

1. Allocate a FillStruct
2. Set the flags in the FillStruct\Flags.w
3. Fill the FillStruct with either integer values or the addresses of our string arguments.
4. Call FillRexxMsg with the LONG address of our rexxmsg and the LONG address of our FillStruct.

To accomplish this takes the following code:

```
;Allocate our FillStruct (called F)
DEFTYPE.FillStruct F
;assign some string arguments
T$="open":T1$="-0123456789"
;Fill in our FillStruct with flags and (&) addresses of our strings
F\Flags=%0010000000000000,&T$,&T1$,4
;Third argument here is an integer (4).
Port.l=CreateMsgPort("host")
msg.l=CreateRexxMsg(Port,"vc","host")
FillRexxMsg msg,&F
;<-3 ergs see #RXFUNC
SendRexxCommand msg,"",#RXFUNCI #RXFF RESULTI 3
```

CreateArgString("this is a string")

Builds an ARexx compatible argstring structure around the provided string. All strings sent to, or received from ARexx are in the form of argstrings. See the TYPE RexxARG.

If all is well the return will be a LONG address of the argString structure. The pointer will actually point to the NULL terminated String with the remainder of the structure available at negative offsets.

DeleteArgString argstring

Designed to Delete argstrings allocated by either Blitz or ARexx in a system friendly way. It takes only one argument the LONGWORD address of an argstring as returned by CreateArgString().

SendRexxCommand rexxmsg,"commands/ring",#RXCOMMI #RXFF_RESULT

Designed to fill and send a RexxMsg structure to ARexx inorder to get ARexx to do something on your behalf. The arguments are as follows; rexxmsg, the LONGWORD address of a RexxMsg structure as returned by CreateRexxMsg().

commands/ring: the command string you wish to send to ARexx. This is a string as in "this is a string" and will vary depending on what you wish to do with ARexx. Normally this will be the name of an ARexx script file you wish to execute. ARexx will then look for the script by the name as well as the name with the exten added. (This is the exten you used when you created the RexxMsg structure using CreateRexxMsg()). This could also be a string file. That is a complete ARexx script in a single line.

actioncodes: the flag values you use to tell ARexx what you want it to do with the commandstring you have supplied.

COMMAND (ACTION) CODES

The command codes that are currently implemented in the resident process are described below. Commands are listed by their mnemonic codes, followed by the valid modifier flags. The final code value is always the logical OR of the code value and all of the modifier flags selected. The command code is installed in the rm_Action field of the message packet.

RXADDCON:

This code specifies an entry to be added to the clip list. Parameter slot ARG0 points to the name string, slot ARG1 points to the value string, and slot ARG2 contains the length of the value string.

The name and value arguments do not need to be argstrings, but can be just pointers to storage areas. The name should be a null-terminated string, but the value can contain arbitrary data including nulls.

RXADDFH:

This action code specifies a function host to be added to the library list. Parameter slot ARG0 points to the (null-terminated) host name string, and slot ARG1 holds the search priority for the node. The search priority should be an integer between 100 and -100 inclusive, the remaining priority ranges are reserved for future extensions. If a node already exists with the same name, the packet is returned with a warning level error code.

Note that no test is made at this time as to whether the host port exists.

RXADDLIB:

This code specifies an entry to be added to the library list. Parameter slot ARG0 points to a null-terminated name string referring either to a function library or a function host. Slot ARG1 is the priority for the node andshould be an integer between 100 and -100 inclusive; the remaining priority ranges are reserved for future extensions. Slot ARG2 contains the entry Point offset and slot ARG3 is the library version number. If a node already exists with the same name,the packet is returned with a warning level error code. Otherwise,a new entry is added and the library or host becomes available to ARexx programs. Note that no test is made at this time as to whether the library exists and can be opened.

RXCOMM [RXFF_TOKEN] [RXFF STRING] [RXFF_RESULT] [RXFF NOIO]

Specifies a command-mode invocation of an ARexx program. Parameter slot ARG0 must contain an argstring Pointer to the command string. The RXFB TOKEN flag specifies that the command line is to be tokenised before being passed to the invoked program. The RXFB_STRING flag bit indicates that the command string is a “string file.” Command invocations do not normally return result strings, but the RXFB_RESULT flag can be set if the caller is prepared to handle the cleanup associated with a returned string. The RXFB_N010 modifier suppresses the inheritance of the host's input and output streams.

RXFUNC [RXFF_RESULT] [RXFF STRING] [RXFF_NOIO] argcount

This command code specifies a function invoction. Parameter slot ARG0 contains a pointer to the function name string, and slots ARG1 through ARG15 point to the argument strings, all of which must be passed as argstrings. The lower byte of the command code is the argument count, this count excludes the function name string itself. Function calls normally set the RXFB_RESULT flag,but this is not mandatory. The RXFB_STRING modifier indicates that the function name string is actually a “string file”. The RXFB_N010 modifier suppresses the inheritance of the host's input and output streams.

RXREMCON:

This code requests that an entry be removed from the clip list. Parameter slot ARG0 points to the null-terminated name to be removed. The clip list is searched for a node matching the supplied name, and if a match is tound the list node is removed and recycled. If no match is found the packet is returned with a warning error code.

RXREMLIB:

This command removes a library list entry. Parameter slot ARG0 points to the null terminated string specifying the library to be removed. The library list is searched for a node matching the library name,and if a match is found the node is removed and released. If no match is found the packet is returned with a warning error code. The libary node will not be removed if the library is currently being used by an ARexx program.

RXTCCLS:

This code requests that the global tracing console be closed. The console window will be closed immediately unless one or more ARexx programs are waiting for input from the console. In this event, the window will be closed as soon as the active programs are no longer using it.

RXTCOPN:

This command requests that the global tracing console be opened. Once the console is open, all active ARexx programs will divert their tracing output to the console. Tracing input (for interactive debugging)will also be diverted to the new console. Only one console can be opened; subsequent RXTCOPN requests will be returned with a warning error message.

MODIFIER FLAGS

Command codes may include modifier flags to select various processing options. Modifier flags are specific to certain commands,and are ignored otherwise.

RXFF_NOIO:

This modifier is used with the RXCOMM and RXFUNC command codes to suppress the automatic inheritance of the host's input and output streams.

RXFF NONRET:

Specifies that the message packet is to be recycled by the resident process rather than being returned to the sender. This implies that the sender doesn’t care about whether the requested action succeeded,since the returned packet provides the only means of acknowledgement. (RXFF_NONRET MUST NOT BE USED AT ANY TIME)

RXFF RESULT:

This modifer is valid with the RXCOMM and RXFUNC commands, and requests that the called program return a result string. If the program EXITs (or RETURNs)with an expression, the expression result is returned to the caller as an argstring. This argstring then becomes the callers responsibility to release. This is automatically accomplished by using GetResultString(). It is therefore imperitive that if you use RXFF_RESULT then you must use GetResultString() when the message packet is returned to you or you will incure a memory loss equal to the size of the argstring Structure.

RXFF_STRING:

This modifer is valid with the RXCOMM and RXFUNC command codes. It indicates that command or function argument (in slot ARG0) is a “string file” rather than a file name.

RXFF_TOKEN:

This flag is used with the RXCOMM code to request that the command string be completely tokenised before being passed to the invoked program. Programs invoked as commands normally have only a single argument string. The tokenization process uses “white space” to separate the tokens, except within quoted strings. Quoted strings can use either single or double quotes, and the end of the command string (a null character) is considered as an implicit closing quote.

ReplyRexxMsg replyrexxmsg rexxmsg,result1,result2,“resultstring”

When ARexx sends you a RexxMsg (Other than a reply to yours ie. sending yours back to you with results) you must repl to the message before ARexx will continue or free that memory associated with that rexxmsg. ReplyRexxMsg accomplishes this for you. ReplyRexxMsg also will only reply to message that requires a reply so you do not have to include message checking routines in your source simply call ReplyRexxMsg on every message you receive wether it is a command or not.

The arguments are:

rexxmsg is the LONGWORD address of the RexxMsg Arexx sent you as returned by GetMsg_(Port).

Result1 is 0 or a severity value if there was an error.

Result2 is 0 or an Arexx error number if there was an error processing the command that was contained in the message.

Resultstring is the result string to be sent back to Arexx. This will only be sent if Arexx requested one and Resultl and 2 are 0.

```
ReplyRexxMsg rexxmsg,0,0,“THE RETURNED MESSAGE”
```

GetRexxResult() result.l=GetRexxResult(rexxmsg,resultnum)

Extracts either of the two result numbers from the RexxMsg structure. Care must be taken with this Function to ascertain wether you are dealing with error codes or a resultstring address. Basically if result 1 is zero then result 2 will either be zero or contain an argstring pointer to the resultstring. This should then be obtained using GetResultString().

The arguments to GetRexxResult are:

rexxmsg is the LONGWORD address of a RexxMsg structure returned from ARexx.

Resultnum is either 1 or 2 depending on wether you wish to check result 1 or result 2.

GetRexxCommand(rexxmsg,argnum)

Allows you access to all 16 argstring slots in the given RexxMsg. Slot 1 contains the command string sent by ARexx in a command message so this allows you to extract the command.

The arguments are:

rexxmsg is a LONGWORD address of the rexxmsg structure as returned by RexxEvent()

afgnum is an integer from 1 to 16 specifying the argstring slot you wish to get an argstring from. YOU MUST KNOW THAT THERE IS AN ARGSTRING THERE.

GetResultString(rexxmag)

Allows you to extract the result string returned to you by ARexx after it has completed the action you requested. ARexx will only send back a result string if you asked for one (using the actioncodes) and the requested action was successful.

Wait

Halts all program execution until an event occurs that the program is interested in. Any Intuition event such as clicking on a gadget in a window will start program execution again.

A message arriving at a msgport will also start program execution again. So you may use Wait to wait for input from any source including messages from ARexx to your program.

Wait should always be paired with EVENT if you need to consider Intuition events in your event handler loop.

RexxEvent(port)

Our ARexx equivalent of EVENT(). Its purpose is to check the given port to see if there is a message waiting there for us. It should be called atter a WAIT and will either return a NULL to us if there was no message or the LONG address of a rexxmsg structure if there was a message waiting. Multiple ARexx msgports can be handled using separate calls to RexxEvent():

```
Wait
Rmsg1.l=RexxEvent(Port1)
Rmsg2.l=RexxEvent(Port2)
```

RexxEvent also takes care of automatically clearing the rexxmsg if it is our message being returned to us.

The argument is the LONG address of a MsgPort as returned by CreateMsgPort().

IsRexxMsg(rexxmsg)

Tests the argument (a LONGWORD pointer hopefully to a message packet) to see if it is a RexxMsg Packet. If it is TRUE is returned (1) or FALSE if it is not (0). As the test is non destructive and extensive passing a NULL value or a LONGWORD that does not point to a message structure (Intuition or ARexx) will return as FALSE.

RexxError() errorstring$=RexxError(errorcode)

Converts a numerical error code such as you would get from GetRexxResult (msg2) into an understandable string error message. If the errorcode is not known to ARexx a string stating so is returned ensuring that this function will always succeed.

R-29: BREXX

The Blitz BRexx commands allow you to take control of certain aspects of Intuition. Through BRexx, your programs can ‘fool’ Intuition into thinking that the mouse has been played with, or the keyboard has been used. This is ideal for giving the ability to perform ‘macros’ - where one keystroke can set off a chain of pre-defined events.

The BRexx commands support tape objects. These are predefined sequences of events which may be played back at any time. The convenient Record command can be used to easily create tapes. Using the MacroKey command, tapes may also be attached to any keystroke to be played back instantly at the push of a button!

Please note that none of the BRexx commands are available in Blitz mode.

AbsMouse x,y

Allows you to position the mouse pointer at an absolute display location. The x parameter specifies how far across the display the pointer is to be positioned, while the y parameter specifies how far down the display. X must be in the range zero through 639. Y must be in the range zero through 399 for NTSC machines, or zero through 511 for PAL machines.

RelMouse xoffset,yoffset

Allows you to move the mouse pointer a relative distance from its current location. Positive offset parameters will move the pointer rightwards and downwards, while negative offset parameters will move the pointer leftwards and upwards.

MouseButton button,On|Off

Allows you to alter the status of the Amiga’s left or right mouse buttons. Button should be set to zero to alter the left mouse button, or one to alter the right mouse button. On/Off refers to whether the mouse button should be pressed (On) or released (Off).

ClickButton button

Identical to executing two MouseButton commands - one for pressing the mouse button down, and one for releasing it. This can be used for such things as gadget selection.

Type string$

Causes Intution to behave exactly as if a certain series of keyboard characters had been entered. These are normally sent to the currently active window.

Record [tape#]

Allows you to create a tape object. Tape objects are sequences of mouse and/or keyboard events which may be played back at any time. When a tape# parameter is supplied to the Record command, recording will begin. From that point on, all mouse and keyboard activity will be recorded onto the specified tape. The Record command with no parameters will cause any recording to finish.

PlayBack [tape#]

Begins playback of a previously created tape object. When a tape# parameter is supplied, playback of the specified tape will commence. If no parameter is supplied, any tape which may be in the process of being played back will finish.

QuickPlay On|Off

Will alter the way tapes are played using the PlayBack command. If QuickPlay is enabled by use of an On parameter, then all PlayBack commands will cause tapes to be played with no delays between actions. This means any pauses which may be present in a tape (for instance, delays between mouse movements) will be ignored when it is played back. QuickPlay Off will return PlayBack to its default mode of including all tape pauses. This is sometimes necessary when playing back tapes which must at some point wait for disk access to finish before continuing.

PlayWait

May be used to halt program flow until a PlayBack of a tape has finished.

XStatus

Returns a value depending upon the current state of the BRexx system. Possible return values and their meanings are as follows:

0	BRexx is currently inactive. No tapes are being recorded or played back.
1	BRexx is currently in the process of recording a tape. This may be due to either the Record or TapeTrap commands.
2	BRexx is currently playing a tape back.

SaveTape tape#,filename$

Allows you to save a previously created tape object out to disk. This tape may later be reloaded using LoadTape.

LoadTape tape#,filename$

Allows you to load a tape object previously saved with SaveTape for use with the PlayBack command.

TapeTrap [tape#]

Allows you to record a sequence of AbsMouse, RelMouse, MouseButton and ClickButton events to a tape object. TapeTrap works similarly to Record, in that both commands are used to create a tape. However, whereas Record receives information from the actual mouse and keyboard, TapeTrap receives information from any AbsMouse, RelMouse, MouseButton and ClickButton commands which may be executed. TapeTrap with no parameter will finish tape creation.

QuietTrap On|Off

Determines the way in which any TapeTrapping will be executed. QuietTrap On will cause any AbsMouse, RelMouse, MouseButton and ClickButton commands to be recorded to tape, but not to actually have any effect on the porgram currently running.

QuietTrap Off will cause any AbsMouse, RelMouse, MouseButton and ClickButton commands to be recorded to tape, AND to cause their usual effects. QuietTrap Off is the default mode.

MacroKey tape#,rawkey,qualifier

Causes a previously defined tape object to be attached to a particular keyboard key. Rawkey and qualifier define the key the tape should be attached to.

FreeMacroKey rawkey,qualifier

Causes a previously defined macro key to be removed so that a BRexx tape is no longer attached to it.

R-30: SERIAL PORT

The following are a set of commands to drive both the single RS232 serial port on an Amiga as well as supporting multiserial port cards such as the A2232 card. The unit# in the following commands should be set to 0 for the standard RS232 port, unit 1 refers to the default serial port set by the advanced serial preferences program and unit 2 refer to any extra serial ports available.

OpenSerial device$,unit#,baud,io_serflags

Used to configure a serial port for use. As with OpenFile, OpenSerial is a function and returns zero if it fails. If it succeeds advanced users may note the return result is the location of the IOExtSer structure. The device$ should be "serial.device" or compatible device driver. The baud rate should be in the range of 110-292,000. The io_serflags parameter can include the following flags:

#serf_xdisabled	128	Disable xon/xoff
#serf_eofmode	64	Enable eof checking
#serf_shared	32	Set if you don't need exclusive use of port
#serf_rad_boogie	16	High speed mode
#serf_queuedbrk	8	If set a break command waits for buffer empty
#serf_7wire	4	If set use 7 wire RS232
#serf_parity_odd	2	Select odd parity (even if not set)
#serf_parity_on	1	Enable parity checking

WriteSerial unit#,byte

Sends one byte to the serial port. Unit# defines which serial port is used. If you are sending characters use the Asc() function to convert the character to a byte e.g.

```
WriteSerial 0,asc("b").
```

WriteSerialString unit#,string

Similar to WriteSerial but sends a complete string to the serial port.

ReadSerial(unit#)

Returns the next byte waiting in the serial port’s read buffer. If the buffer is empty it returns -1. It is best to use a word type (var.w=ReadSerial(0)) as a byte will not be able to differentiate between -1 and 255.

ReadSerialString(unit#)

Puts the serial port's read buffer into a string, if the buffer is empty the function will return a null string (length=0).

CloseSerial unit#

Closes the port, enabling other programs to use it.

Note: Blitz will automatically close all ports that are opened when a program ends.

SetSerialBuffer unit#,bufferlength

Changes the size of the ports read buffer. This may be useful if your program is not always handling serial port data or is receiving and processing large chunks of data. The smallest size for the internal serial port (unit#0) is 64 bytes. The bufferlength variable is in bytes.

SetSerialLens unit#,readlen,writelen,stopbits

Allows you to change the size of characters read and written by the serial device. Generally readlen=writelen and should be set to either 7 or 8, stopbits should be set to 1 or 2. Default values are 8,8,1.

SetSerialParams unit#

For advanced users, SetSerialParams tells the serial port when parameters are changed. This would only be necesary if they were changed by poking offsets from IOExtSer which is returned by the OpenSerial command.

SerialEvent(unit#)

Used when your program is handling events from more than one source, Windows, ARexx etc. This command is currently not implemented.

ReadSerialMem unit#,address,length

Will fill the given memory space with data from the given serial port.

WriteSerialMem unit#,address,length

Sends out the given memory space out the given serial port.

APPENDIX 1: COMPILE TIME ERRORS

The following is a list of all the Blitz 2 compile time errors. Blitz 2 will print these messages when unable to compile a line of your code and fails. The cursor will be placed on the line with the offending error in most cases.

Sometimes the cause of the error will not be directly related to where Blitz 2 ceased compiling. Any reference to an include file or a macro could mean the error is there and not on the line referenced.

General Syntax Errors

Syntax Error: Check for typing mistakes and check your syntax with the reference manual.

Garbage at End of Line: A syntax error of sorts. Causes are usually typos and missing semi colons from the beginning of Remarks. Also a .type suffix when accessing NewType items will generate this error.

Numeric Over Flow: The signed value is too large to fit in the variable space provided, if you need bytes to hold 0...255 rather than -128...127 etc turn off Overflow checking in the runtime errors section of the Options requester.

Bad Data: The values following the Data.type statement are not of the same type as precedes the Data statement.

Procedure Related Errors

Not Enough Parameters: The command, statement or function needs more parameters. Use the HELP key for correct number and meaning of parameters with Blitz 2 commands and check Statement and Function definitions in your code.

Duplicate Parameter Variable: Parmaters listed in statements and functions must be unique.

Too Many Parameters: The statement or function was defined needing less parameters than supplied by the calling routine.

Illegal Parameter Type: NewTypes cannot be passed to procedures.

Illegal Procedure Return: The statement or function return is syntactically incorrect.

Illegal End Procedure: The statement or function end is syntactically incorrect.

Shared outside of Procedure: Shared variables are only applicable to procedures. Variable already Shared: Shared variables must be unique in name.

Can't Nest Procedures: Procedures may NOT be defined within procedures, only from the primary code.

Can't Dim Globals in Procedures: Global arrays may be only defined from the pumary code.

Can't Goto/Gosub a Procedure: Goto and Gosub must always point to an existing part of the primary code.

Duplicate Procedure name: A procedure (statement or function) of the same name has been defined previously in the source.

Procedure Not Found: The statement or function has not previously been defined in the source code.

Unterminated Procedure: The End Function or End Statement commands must terminate a procedure definition.

Illegal Procedure Call: The statement or function call is syntactically incorrect.

Illegal Local Name: Not a valid variable name.

Constants Related Errors

Can't Assign Constant: Constant values can only be assigned to constants, no variables please.

Constant Not Defined: A constant (such as #num) has been used in an expression without first being defined.

Constant Already Defined: Constants can only be defined once, i.e. cannot change their value through the code.

Illegal Constant: Same as Can't Assign Constant.

Fractions Not allowed in Constants: Blitz 2 constants can only contain absolute values, they are usually rounded and no error is generated.

Can't Use Constant: Caused by a clash in constant name definitions.

Constant Not Found: Constant has not been defined previously in the source code.

Illegal Constant Expression: A constant may only hold whole numbers, either a decimal place, text or a variable name has been included in the constant definition.

Expression Evaluation Errors

Can’t Assign Expression: The expression cannot be evaluated or the evaluation has generated a value that is incompatible with the equate.

No Terminating Quote: Any text assigns should start and end with quotes.

Precedence Stack Overflow: You have attained an unprecedented level of complexity in your expression and the Blitz 2 evaluation stack has overflowed. A rare beast indeed!

Illegal Errors

Illegal Trap Vector: The 68000 has only 16 trap vectors.

Illegal Immediate Value: An immediate value must be a constant and must be in range. See the 68000 appendix for immediate value ranges.

Illegal Absolute: The Absolute location specified must be defined and in range.

Illegal Displacement: Displacement location specified must be defined and in range.

Illegal Assembler Instruction Size: The Intstruction size is not available, refer to the 68000 appendix for relevant instruction sizes.

Illegal Assembler Addressing Mode: The addressing mode is not available for that opcode, refer to the 68000 appendix for relevant addressing modes.

Library Based Errors

Illegal TokeJsr Token Number: Blitz 2 cannot find the library routine referred to by the TokeJsr command, usually caused by the library not being included in DefLibs, not present in the BlitzLibs: directory or the calculation being wrong (token number = libnumber* 128 + token offset).

Library not Found: 'library number': Blitz2 cannot find the library routine referred to by a Token, usually caused by the library not being'included in DeflLibs or the library not present in the BlitzLibs: directories.

Token Not Found: 'token number': When loading source, Blitz 2 replaces any unfound tokens with ?????, compiling your code with these unknown tokens present will generate the above error.

Include Errors

Already Included: Same source code has already been included previously in the code.

Can't open Include: Blitz 2 cannot find the include file, check the pathname.

Error Reading File: DOS has generated an error during an include.

Program Flow Based Errors

Illegal Else in While Block: See the reference section for the correct use of the Else command with While...Wend blocks.

Until without Repeat: Repeat...Until is a block directive and both must be present.

Repeat Block too large: A Repeat...Until block is limited to 32000 bytes in length.

Repeat Without Until: Repeat...Until is a block directive and both must be present.

If Block Too Large: Blitz2 has a 32K limit for any blocks of code such as IF...ENDIF.

If Without End If: The IF statement has two forms, if the THEN statement is not present then and END IF statment must be present to specify the end of the block.

Duplicate For...Next Error: The same variable has been used for a For...Next loop that is nested within another For...Next loop.

Bad Type For For...Next: The For...Next variable must be of numeric type.

Next Without For: For...Next is a block directive and both commands must be present.

For...Next Block too Long: Blitz2 restricts all blocks of code to 32K in size.

For Without Next: For...Next is a block directive and both commands must be present.

Type Based Errors

Can't Exchange different Lopes: The Exchange command can only swap two variables of the same type.

Can't Exchange NewTypes: Exchange command can't handle NewTypes at present.

Type Too Big: The unsigned value is too large to fit in the variable space provided.

Mismatched Types: Caused by mixing different types illegaly in an evaluation.

Type Mismatch: Same as Mismatched Types.

Can't Compare Types: Some Types are incompatible with operations such as compares.

Can't Convert Types: The two Types are incompatible and one can not be converted to the other.

Duplicate Offset (Entry) Error: The NewType has two entries of the same name.

Duplicated Type: A Type already exists with the same name.

End NewType without NewType: The NEWTYPE ...End NewType is a block directive and both must be present.

Type Not Found: No Type definition exists for the type referred to.

Illegal Type: Not a legal type for that function or statement.

Offset Not Found: The offset has not been defined in the NewType definition.

Element Isn't A Pointer: The variable used is not a *var type and so cannot point to another variable.

Illegal Operator For Type: The operator is not suited for the type used.

Too Many Comma's In Let: The NewType has less entries than the number of values listed after the Let.

Can't Use Comma In Let: The variable you are assigning multiple values is either not a NewType and cannot hold multiple values or the NewType has only one entry.

Illegal Function Type: A function may not return a NEWTYPE .

Conditional Compiling Errors

CNIF/CSIF Without CEND: CNIF and CSIF are block directives and a CEND must conclude the block.

CEND Without CNIF/CSIF...: CNIF...CEND is a block directive and both commands must be present.

Resident Based Errors

Clash In Residents: Residents being very unique animals, must not include the same Macro and Constant definitions.

Can't Load Resident: Blitz2 cannot find the Resident file listed in the Options requester. Check the pathname.

Macro Based Errors

Macro Buffer Overflow: The Options requester in the Blitz2 menu contains a macro buffer size, increase if this error is ever reported. May also be caused by a recursive macro call which generates endless code.

Macro already Defined: Another macro with the same name has already been defined, may have been defined in one of the included resident files as well as somewhere else in the source code.

Can't Create Macro Inside Macro: Macro definitions must occur in the primary code.

Macro Without End Macro: End Macro must end a Macro definition.

Macro Too Big: Macro's are limited to the buffer sizes defined in the Options requester.

Macros Nested Too Deep: Eight levels of macro nesting is available in Blitz2. Should never happen!

Macro Not Found: The macro has not been defined previous to the !macroname{} call.

Array Errors

Illegal Array Type: Should never happen.

Array Not Found: A variable name followed by parenthises has not been previously defined as an array. Other possible mistakes may be the use of brackets instead of curly brackets for macro and procedure calls, Blitz2 thinking instead you are referring to an array name.

Array Is Not A List: A List function has been used on an array that was not dimensioned as a List Array.

Illegal Number Of Dimensions: List arrays are limited to single dimensions.

Array Already Dim'd: An array may not be re-dimensioned.

Can't Create Variable Inside Dim: An undefined variable has been used for a dimension parameter with the Dim statement.

Array Not Yet Dim'd: See Array not found.

Array Not Dim'd: See Array not found.

Interrupt Based Errors

End SetInt Without SetInt: Setlnt...End SetInt is a block directive and both commands must be present.

SetInt Without End SetInt: SetInt...End SetInt is a block directive and both commands must be present.

Can't Use SeVClrInt In Local Mode: Error handling can only be defined by the primary code.

SetErr Not Allowed In Procedures: Error handling can only be defined by the primary code.

Can't use SeVClrlat in Local Mode: Error handling can only be defined by the primary code.

End Setlnt without SetInt: SetInt...End SetInt is a block directive and both commands must be present.

Setlut without End SetInt: Setlnt...End SetInt is a block directive and both commands must be present.

Illegally nested Interrupts: Interrupt handlers can obviously not be nested.

Can't nest SetErr: Interrupt handlers can obviously not be nested.

End SetErr without SetErr: SetErr...End SetErr is a block directive and both must be present.

Illegal Interrupt Number: Amiga interrupts are limited from 0 to 13. These interrupts are listed in the Amiga Hardware reference appendix.

Label Errors Label reference out of context : Should never happen

Label has been used as a Constant: Labels and constants cannot share same name.

Illegal Label Name: Refer to the Programming in Blitz2 chapter for correct variable nomenclature.

Duplicate Label: A label has been defined twice in the same source code. May also occur with macros where a label is not preceded by a \@.

Label not Found: The label has not been defined anywhere in the source code.

Can't Access Label: The label has not been defined in the source code.

Direct Mode Errors

Cont Option Disabled: The Enable Continue option in the Runtime errors of the Options menu has been disabled.

Cont only Available in Direct Mode: Cont can not be called from your code only from the direct mode window.

Library not Available in Direct Mode: Library is only available from within the code.

Illegal direct mode command: Direct mode is unable to execute command entered.

Direct Mode Buffer Overflow: The Options menu contains sizes of all buffers, if make smallest code is in effect extra buffer memory will not be available for direct mode.

Can't Create in Direct Mode: Variables cannot be created using direct mode, only ones defined by your code are available.

Select... End Select Errors

Select without End Select: Select is a block directive and an End Select must conclude the block.

End Select without Select: Select...End Select is a block directive and both must be present.

Default without Select: The Default command is only relevant to the Select...End Select block directive.

Previous Case Block too Large: A Case section in a Select block is larger than 32K. Case Without Select: The Case command is only relevant to the Select...End Select block directive.

Blitz Mode Errors

Only Available in Blitz mode: The command is only available in Blitz mode, refer to the reference section for Blitz/Amiga valid commands.

Only Available in Amiga mode: The command is only available in Amiga mode, refer to the reference section for Blitz/Amiga valid commands.

Strange Beast Errors

Optimiser Error! - $': This should never happen. Please report.

Expression too Complex: Should never happen. Contact Mark directly.

Not Supported: Should never happen.

Illegal Token: Should never happen.

APPENDIX 2: OPERATING SYSTEM CALLS

BLITZLIBS:AMIGALIBS currently supports the EXEC, DOS, GRAPHICS, INTUITION and DISKFONT Amiga libraries.

Each call may be treated as either a command or a function. Functions will always return a long either containing true or false (signifying if the command was successful or failed) or a value relevant to the routine.

When using library calls an underscore character (_) should follow the name. For example:

```
If (dosbase.l=OpenLibrary_("reqtools.library",0))
  ; program execution continues here
  CloseLibrary_(dosbase)
Else
  NPrint "ERROR: Unable to open reqtools.library"
EndIf
```

An asterisk (*) preceding routine names specifies that the calls are private and should not be called from Blitz 2.

EXEC

Supervisor(userFunction)(a5)
module creation
InitCode(startClass,version)(d0/d1)
InitStruct(initTable,memory,size)(a1/a2,d0)
MakeLibrary(funcInit,structInit,libInit,dataSize,segList)(a0/a1/a2,d0/d1)
MakeFunctions(target,functionArray,funcDispBase)(a0/a1/a2)
FindResident(name)(a1)
InitResident(resident,segList)(a1,d1)
diagnostics
Alert(alertNum)(d7)
Debug(flags)(d0)
interrupts
Disable()()
Enable()()
Forbid()()
Permit()()
SetSR(newSR,mask)(d0/d1)
SuperState()()
UserState(sysStack)(d0)
SetIntVector(intNumber,interrupt)(d0/a1)
AddIntServer(intNumber,interrupt)(d0/a1)
RemIntServer(intNumber,interrupt)(d0/a1)
Cause(interrupt)(a1)

memory allocation
Allocate(freeList,byteSize)(a0,d0)
Deallocate(freeList,memoryBlock,byteSize)(a0/a1,d0)
AllocMem(byteSize,requirements)(d0/d1)
AllocAbs(byteSize,location)(d0/a1)
FreeMem(memoryBlock,byteSize)(a1,d0)
AvailMem(requirements)(d1)
AllocEntry(entry)(a0)
FreeEntry(entry)(a0)
lists
Insert(list,node,pred)(a0/a1/a2)
AddHead(list,node)(a0/a1)
AddTail(list,node)(a0/a1)
Remove(node)(a1)
RemHead(list)(a0)
RemTail(list)(a0)
Enqueue(list,node)(a0/a1)
FindName(list,name)(a0/a1)
tasks
AddTask(task,initPC,finalPC)(a1/a2/a3)
RemTask(task)(a1)
FindTask(name)(a1)
SetTaskPri(task,priority)(a1,d0)
SetSignal(newSignals,signalSet)(d0/d1)
SetExcept(newSignals,signalSet)(d0/d1)
Wait(signalSet)(d0)
Signal(task,signalSet)(a1,d0)
AllocSignal(signalNum)(d0)
FreeSignal(signalNum)(d0)
AllocTrap(trapNum)(d0)
FreeTrap(trapNum)(d0)
messages
AddPort(port)(a1)
RemPort(port)(a1)
PutMsg(port,message)(a0/a1)
GetMsg(port)(a0)
ReplyMsg(message)(a1)
WaitPort(port)(a0)
FindPort(name)(a1)
libraries
AddLibrary(library)(a1)
RemLibrary(library)(a1)
OldOpenLibrary(libName)(a1)
CloseLibrary(library)(a1)
SetFunction(library,funcOffset,newFunction)(a1,a0,d0)
SumLibrary(library)(a1)

devices
AddDevice(device)(a1)
RemDevice(device)(a1)
OpenDevice(devName,unit,ioRequest,flags)(a0,d0/a1,d1)
CloseDevice(ioRequest)(a1)
DoIO(ioRequest)(a1)
SendIO(ioRequest)(a1)
CheckIO(ioRequest)(a1)
WaitIO(ioRequest)(a1)
AbortIO(ioRequest)(a1)
resources
AddResource(resource)(a1)
RemResource(resource)(a1)
OpenResource(resName)(a1)
misc
RawDoFmt(formatString,dataStream,putChProc,putChData)(a0/a1/a2/a3)
GetCC()()
TypeOfMem(address)(a1)
Procure(semaport,bidMsg)(a0/a1)
Vacate(semaport)(a0)
OpenLibrary(libName,version)(a1,d0)
functions in v33 or higher (distributed as Release 1.2)
signal semaphores (note funny registers)
InitSemaphore(sigSem)(a0)
ObtainSemaphore(sigSem)(a0)
ReleaseSemaphore(sigSem)(a0)
AttemptSemaphore(sigSem)(a0)
ObtainSemaphoreList(sigSem)(a0)
ReleaseSemaphoreList(sigSem)(a0)
FindSemaphore(sigSem)(a1)
AddSemaphore(sigSem)(a1)
RemSemaphore(sigSem)(a1)
kickmem support
SumKickData()()
more memory support
AddMemList(size,attributes,pri,base,name)(d0/d1/d2/a0/a1)
CopyMem(source,dest,size)(a0/a1,d0)
CopyMemQuick(source,dest,size)(a0/a1,d0)
cache
functions in v36 or higher (distributed as Release 2.0)
CacheClearU()()
CacheClearE(address,length,caches)(a0,d0/d1)
CacheControl(cacheBits,cacheMask)(d0/d1)
misc
CreateIORequest(port,size)(a0,d0)
DeleteIORequest(iorequest)(a0)
CreateMsgPort()()
DeleteMsgPort(port)(a0)
ObtainSemaphoreShared(sigSem)(a0)

even more memory support
AllocVec(byteSize,requirements)(d0/d1)
FreeVec(memoryBlock)(a1)
CreatePrivatePool(requirements,puddleSize,puddleThresh)(d0/d1/d2)
DeletePrivatePool(poolHeader)(a0)
AllocPooled(memSize,poolHeader)(d0/a0)
FreePooled(memory,poolHeader)(a1,a0)
misc
AttemptSemaphoreShared(sigSem)(a0)
ColdReboot()()
StackSwap(newStack)(a0)
task trees
ChildFree(tid)(d0)
ChildOrphan(tid)(d0)
ChildStatus(tid)(d0)
ChildWait(tid)(d0)
future expansion
CachePreDMA(address,length,flags)(a0/a1,d1)
CachePostDMA(address,length,flags)(a0/a1,d1)

DOS

Open(name,accessMode)(d1/d2)
Close(file)(d1)
Read(file,buffer,length)(d1/d2/d3)
Write(file,buffer,length)(d1/d2/d3)
Input()()
Output()()
Seek(file,position,offset)(d1/d2/d3)
DeleteFile(name)(d1)
Rename(oldName,newName)(d1/d2)
Lock(name,type)(d1/d2)
UnLock(lock)(d1)
DupLock(lock)(d1)
Examine(lock,fileInfoBlock)(d1/d2)
ExNext(lock,fileInfoBlock)(d1/d2)
Info(lock,parameterBlock)(d1/d2)
CreateDir(name)(d1)
CurrentDir(lock)(d1)
IoErr()()
CreateProc(name,pri,segList,stackSize)(d1/d2/d3/d4)
Exit(returnCode)(d1)
LoadSeg(name)(d1)
UnLoadSeg(seglist)(d1)
DeviceProc(name)(d1)
SetComment(name,comment)(d1/d2)
SetProtection(name,protect)(d1/d2)
DateStamp(date)(d1)
Delay(timeout)(d1)
WaitForChar(file,timeout)(d1/d2)
ParentDir(lock)(d1)
IsInteractive(file)(d1)

Execute(string,file,file2)(d1/d2/d3)
functions in v36 or higher (distributed as Release 2.0)
DOS object creation/deletion
AllocDosObject(type,tags)(d1/d2)
FreeDosObject(type,ptr)(d1/d2)
packet level routines
DoPkt(port,action,arg1,arg2,arg3,arg4,arg5)(d1/d2/d3/d4/d5/d6/d7)
SendPkt(dp,port,replyport)(d1/d2/d3)
WaitPkt()()
ReplyPkt(dp,res1,res2)(d1/d2/d3)
AbortPkt(port,pkt)(d1/d2)
Record Locking
LockRecord(fh,offset,length,mode,timeout)(d1/d2/d3/d4/d5)
LockRecords(recArray,timeout)(d1/d2)
UnLockRecord(fh,offset,length)(d1/d2/d3)
UnLockRecords(recArray)(d1)
Buffered File I/O
SelectInput(fh)(d1)
SelectOutput(fh)(d1)
FGetC(fh)(d1)
FPutC(fh,ch)(d1/d2)
UnGetC(fh,character)(d1/d2)
FRead(fh,block,blocklen,number)(d1/d2/d3/d4)
FWrite(fh,block,blocklen,number)(d1/d2/d3/d4)
FGets(fh,buf,buflen)(d1/d2/d3)
FPuts(fh,str)(d1/d2)
VFWritef(fh,format,argarray)(d1/d2/d3)
VFPrintf(fh,format,argarray)(d1/d2/d3)
Flush(fh)(d1)
SetVBuf(fh,buff,type,size)(d1/d2/d3/d4)
DOS Object Management
DupLockFromFH(fh)(d1)
OpenFromLock(lock)(d1)
ParentOfFH(fh)(d1)
ExamineFH(fh,fib)(d1/d2)
SetFileDate(name,date)(d1/d2)
NameFromLock(lock,buffer,len)(d1/d2/d3)
NameFromFH(fh,buffer,len)(d1/d2/d3)
SplitName(name,seperator,buf,oldpos,size)(d1/d2/d3/d4/d5)
SameLock(lock1,lock2)(d1/d2)
SetMode(fh,mode)(d1/d2)
ExAll(lock,buffer,size,data,control)(d1/d2/d3/d4/d5)
ReadLink(port,lock,path,buffer,size)(d1/d2/d3/d4/d5)
MakeLink(name,dest,soft)(d1/d2/d3)
ChangeMode(type,fh,newmode)(d1/d2/d3)
SetFileSize(fh,pos,mode)(d1/d2/d3)
Error Handling
SetIoErr(result)(d1)
Fault(code,header,buffer,len)(d1/d2/d3/d4)
PrintFault(code,header)(d1/d2)
ErrorReport(code,type,arg1,device)(d1/d2/d3/d4)

Process Management
Cli()()
CreateNewProc(tags)(d1)
RunCommand(seg,stack,paramptr,paramlen)(d1/d2/d3/d4)
GetConsoleTask()()
SetConsoleTask(task)(d1)
GetFileSysTask()()
SetFileSysTask(task)(d1)
GetArgStr()()
SetArgStr(string)(d1)
FindCliProc(num)(d1)
MaxCli()()
SetCurrentDirName(name)(d1)
GetCurrentDirName(buf,len)(d1/d2)
SetProgramName(name)(d1)
GetProgramName(buf,len)(d1/d2)
SetPrompt(name)(d1)
GetPrompt(buf,len)(d1/d2)
SetProgramDir(lock)(d1)
GetProgramDir()()
Device List Management
SystemTagList(command,tags)(d1/d2)
AssignLock(name,lock)(d1/d2)
AssignLate(name,path)(d1/d2)
AssignPath(name,path)(d1/d2)
AssignAdd(name,lock)(d1/d2)
RemAssignList(name,lock)(d1/d2)
GetDeviceProc(name,dp)(d1/d2)
FreeDeviceProc(dp)(d1)
LockDosList(flags)(d1)
UnLockDosList(flags)(d1)
AttemptLockDosList(flags)(d1)
RemDosEntry(dlist)(d1)
AddDosEntry(dlist)(d1)
FindDosEntry(dlist,name,flags)(d1/d2/d3)
NextDosEntry(dlist,flags)(d1/d2)
MakeDosEntry(name,type)(d1/d2)
FreeDosEntry(dlist)(d1)
IsFileSystem(name)(d1)
Handler Interface
Format(filesystem,volumename,dostype)(d1/d2/d3)
Relabel(drive,newname)(d1/d2)
Inhibit(name,onoff)(d1/d2)
AddBuffers(name,number)(d1/d2)
Date, Time Routines
CompareDates(date1,date2)(d1/d2)
DateToStr(datetime)(d1)
StrToDate(datetime)(d1)

Image Management
InternalLoadSeg(fh,table,funcarray,stack)(d0/a0/a1/a2)
InternalUnLoadSeg(seglist,freefunc)(d1/a1)
NewLoadSeg(file,tags)(d1/d2)
AddSegment(name,seg,system)(d1/d2/d3)
FindSegment(name,seg,system)(d1/d2/d3)
RemSegment(seg)(d1)
Command Support
CheckSignal(mask)(d1)
ReadArgs(template,array,args)(d1/d2/d3)
FindArg(keyword,template)(d1/d2)
ReadItem(name,maxchars,cSource)(d1/d2/d3)
StrToLong(string,value)(d1/d2)
MatchFirst(pat,anchor)(d1/d2)
MatchNext(anchor)(d1)
MatchEnd(anchor)(d1)
ParsePattern(pat,buf,buflen)(d1/d2/d3)
MatchPattern(pat,str)(d1/d2)
FreeArgs(args)(d1)
FilePart(path)(d1)
PathPart(path)(d1)
AddPart(dirname,filename,size)(d1/d2/d3)
Notification
StartNotify(notify)(d1)
EndNotify(notify)(d1)
Environment Variable functions
SetVar(name,buffer,size,flags)(d1/d2/d3/d4)
GetVar(name,buffer,size,flags)(d1/d2/d3/d4)
DeleteVar(name,flags)(d1/d2)
FindVar(name,type)(d1/d2)
CliInitNewcli(dp)(a0)
CliInitRun(dp)(a0)
WriteChars(buf,buflen)(d1/d2)
PutStr(str)(d1)
VPrintf(format,argarray)(d1/d2)
these were unimplemented until dos 36.147
ParsePatternNoCase(pat,buf,buflen)(d1/d2/d3)
MatchPatternNoCase(pat,str)(d1/d2)
this was added for v37 dos, returned 0 before then.
SameDevice(lock1,lock2)(d1/d2)

GRAPHICS

BitMap primitives
BltBitMap(srcBitMap,xSrc,ySrc,destBitMap,xDest,yDest,xSize,ySize,minterm,mask,tempA)(a0,d0/d1/a1,d2/d3/d4/d5/d6/d7/a2)
BltTemplate(source,xSrc,srcMod,destRP,xDest,yDest,xSize,ySize)(a0,d0/d1/a1,d2/d3/d4/d5)
Text routines
ClearEOL(rp)(a1)
ClearScreen(rp)(a1)
TextLength(rp,string,count)(a1,a0,d0)
Text(rp,string,count)(a1,a0,d0)

SetFont(rp,textFont)(a1,a0)
OpenFont(textAttr)(a0)
CloseFont(textFont)(a1)
AskSoftStyle(rp)(a1)
SetSoftStyle(rp,style,enable)(a1,d0/d1)
Gels routines
AddBob(bob,rp)(a0/a1)
AddVSprite(vSprite,rp)(a0/a1)
DoCollision(rp)(a1)
DrawGList(rp,vp)(a1,a0)
InitGels(head,tail,gelsInfo)(a0/a1/a2)
InitMasks(vSprite)(a0)
RemIBob(bob,rp,vp)(a0/a1/a2)
RemVSprite(vSprite)(a0)
SetCollision(num,routine,gelsInfo)(d0/a0/a1)
SortGList(rp)(a1)
AddAnimOb(anOb,anKey,rp)(a0/a1/a2)
Animate(anKey,rp)(a0/a1)
GetGBuffers(anOb,rp,flag)(a0/a1,d0)
InitGMasks(anOb)(a0)
General graphics routines
DrawEllipse(rp,xCenter,yCenter,a,b)(a1,d0/d1/d2/d3)
AreaEllipse(rp,xCenter,yCenter,a,b)(a1,d0/d1/d2/d3)
LoadRGB4(vp,colors,count)(a0/a1,d0)
InitRastPort(rp)(a1)
InitVPort(vp)(a0)
MrgCop(view)(a1)
MakeVPort(view,vp)(a0/a1)
LoadView(view)(a1)
WaitBlit()()
SetRast(rp,pen)(a1,d0)
Move(rp,x,y)(a1,d0/d1)
Draw(rp,x,y)(a1,d0/d1)
AreaMove(rp,x,y)(a1,d0/d1)
AreaDraw(rp,x,y)(a1,d0/d1)
AreaEnd(rp)(a1)
WaitTOF()()
QBlit(blit)(a1)
InitArea(areaInfo,vectorBuffer,maxVectors)(a0/a1,d0)
SetRGB4(vp,index,red,green,blue)(a0,d0/d1/d2/d3)
QBSBlit(blit)(a1)
BltClear(memBlock,byteCount,flags)(a1,d0/d1)
RectFill(rp,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3)
BltPattern(rp,mask,xMin,yMin,xMax,yMax,maskBPR)(a1,a0,d0/d1/d2/d3/d4)
ReadPixel(rp,x,y)(a1,d0/d1)
WritePixel(rp,x,y)(a1,d0/d1)
Flood(rp,mode,x,y)(a1,d2,d0/d1)
PolyDraw(rp,count,polyTable)(a1,d0/a0)
SetAPen(rp,pen)(a1,d0)
SetBPen(rp,pen)(a1,d0)
SetDrMd(rp,drawMode)(a1,d0)

```
InitView(view)(a1)
CBump(copList)(a1)
CMove(copList,destination,data)(a1,d0/d1)
CWait(copList,v,h)(a1,d0/d1)
VBeamPos()()
InitBitMap(bitMap,depth,width,height)(a0,d0/d1/d2)
ScrollRaster(rp,dx,dy,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3/d4/d5)
WaitBOVP(vp)(a0)
GetSprite(sprite,num)(a0,d0)
FreeSprite(num)(d0)
ChangeSprite(vp,sprite,newData)(a0/a1/a2)
MoveSprite(vp,sprite,x,y)(a0/a1,d0/d1)
LockLayerRom(layer)(a5)
UnlockLayerRom(layer)(a5)
SyncSBitMap(layer)(a0)
CopySBitMap(layer)(a0)
OwnBlitter()()
DisownBlitter()()
InitTmpRas(tmpRas,buffer,size)(a0/a1,d0)
AskFont(rp,textAttr)(a1,a0)
AddFont(textFont)(a1)
RemFont(textFont)(a1)
AllocRaster(width,height)(d0/d1)
FreeRaster(p,width,height)(a0,d0/d1)
AndRectRegion(region,rectangle)(a0/a1)
OrRectRegion(region,rectangle)(a0/a1)
NewRegion()()
ClearRectRegion(region,rectangle)(a0/a1)
ClearRegion(region)(a0)
DisposeRegion(region)(a0)
FreeVPortCopLists(vp)(a0)
FreeCopList(copList)(a0)
ClipBlit(srcRP,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,minterm)(a0,d0/d1/a1,d2/d3/d4/d5/d6)
XorRectRegion(region,rectangle)(a0/a1)
FreeCprList(cprList)(a0)
GetColorMap(entries)(d0)
FreeColorMap(colorMap)(a0)
GetRGB4(colorMap,entry)(a0,d0)
ScrollVPort(vp)(a0)
UCopperListInit(uCopList,n)(a0,d0)
FreeGBuffers(anOb,rp,flag)(a0/a1,d0)
BltBitMapRastPort(srcBitMap,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,minterm)(a0,d0/d1/a1,d2/d3/d4/d5/
d6)
OrRegionRegion(srcRegion,destRegion)(a0/a1)
XorRegionRegion(srcRegion,destRegion)(a0/a1)
AndRegionRegion(srcRegion,destRegion)(a0/a1)
SetRGB4CM(colorMap,index,red,green,blue)(a0,d0/d1/d2/d3)
BltMaskBitMapRastPort(srcBitMap,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,minterm,bltMask)(a0,d0/d1/
a1,d2/d3/d4/d5/d6/a2)
AttemptLockLayerRom(layer)(a5)
```

functions in v36 or higher (distributed as Release 2.0)
GfxNew(gfxNodeType)(d0)
GfxFree(gfxNodePtr)(a0)
GfxAssociate(associateNode,gfxNodePtr)(a0/a1)
BitMapScale(bitScaleArgs)(a0)
ScalerDiv(factor,numerator,denominator)(d0/d1/d2)
TextExtent(rp,string,count,textExtent)(a1,a0,d0/a2)
TextFit(rp,string,strLen,textExtent,constrainingExtent,strDirection,constrainingBitWidth,constrainingBitHeight
)(a1,a0,d0/a2/a3,d1/d2/d3)
GfxLookUp(associateNode)(a0)
VideoControl(colorMap,tagarray)(a0/a1)
OpenMonitor(monitorName,displayID)(a1,d0)
CloseMonitor(monitorSpec)(a0)
FindDisplayInfo(displayID)(d0)
NextDisplayInfo(displayID)(d0)
GetDisplayInfoData(handle,buf,size,tagID,displayID)(a0/a1,d0/d1/d2)
FontExtent(font,fontExtent)(a0/a1)
ReadPixelLine8(rp,xstart,ystart,width,array,tempRP)(a0,d0/d1/d2/a2,a1)
WritePixelLine8(rp,xstart,ystart,width,array,tempRP)(a0,d0/d1/d2/a2,a1)
ReadPixelArray8(rp,xstart,ystart,xstop,ystop,array,temprp)(a0,d0/d1/d2/d3/a2,a1)
WritePixelArray8(rp,xstart,ystart,xstop,ystop,array,temprp)(a0,d0/d1/d2/d3/a2,a1)
GetVPModeID(vp)(a0)
ModeNotAvailable(modeID)(d0)
WeighTAMatch(reqTextAttr,targetTextAttr,targetTags)(a0/a1/a2)
EraseRect(rp,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3)
ExtendFont(font,fontTags)(a0/a1)
StripFont(font)(a0)

INTUITION

Public functions OpenIntuition() and Intuition() are intentionally not documented.
OpenIntuition()()
Intuition(iEvent)(a0)
AddGadget(window,gadget,position)(a0/a1,d0)
ClearDMRequest(window)(a0)
ClearMenuStrip(window)(a0)
ClearPointer(window)(a0)
CloseScreen(screen)(a0)
CloseWindow(window)(a0)
CloseWorkBench()()
CurrentTime(seconds,micros)(a0/a1)
DisplayAlert(alertNumber,string,height)(d0/a0,d1)
DisplayBeep(screen)(a0)
DoubleClick(sSeconds,sMicros,cSeconds,cMicros)(d0/d1/d2/d3)
DrawBorder(rp,border,leftOffset,topOffset)(a0/a1,d0/d1)
DrawImage(rp,image,leftOffset,topOffset)(a0/a1,d0/d1)
EndRequest(requester,window)(a0/a1)
GetDefPrefs(preferences,size)(a0,d0)
GetPrefs(preferences,size)(a0,d0)
InitRequester(requester)(a0)
ItemAddress(menuStrip,menuNumber)(a0,d0)
ModifyIDCMP(window,flags)(a0,d0)

ModifyProp(gadget,window,requester,flags,horizPot,vertPot,horizBody,vertBody)(a0/a1/a2,d0/d1/d2/d3/d4)
MoveScreen(screen,dx,dy)(a0,d0/d1)
MoveWindow(window,dx,dy)(a0,d0/d1)
OffGadget(gadget,window,requester)(a0/a1/a2)
OffMenu(window,menuNumber)(a0,d0)
OnGadget(gadget,window,requester)(a0/a1/a2)
OnMenu(window,menuNumber)(a0,d0)
OpenScreen(newScreen)(a0)
OpenWindow(newWindow)(a0)
OpenWorkBench()()
PrintIText(rp,iText,left,top)(a0/a1,d0/d1)
RefreshGadgets(gadgets,window,requester)(a0/a1/a2)
RemoveGadget(window,gadget)(a0/a1)
The official calling sequence for ReportMouse is given below. Note the register order. For the complete story, read the ReportMouse autodoc.
ReportMouse(flag,window)(d0/a0)
Request(requester,window)(a0/a1)
ScreenToBack(screen)(a0)
ScreenToFront(screen)(a0)
SetDMRequest(window,requester)(a0/a1)
SetMenuStrip(window,menu)(a0/a1)
SetPointer(window,pointer,height,width,xOffset,yOffset)(a0/a1,d0/d1/d2/d3)
SetWindowTitles(window,windowTitle,screenTitle)(a0/a1/a2)
ShowTitle(screen,showIt)(a0,d0)
SizeWindow(window,dx,dy)(a0,d0/d1)
ViewAddress()()
ViewPortAddress(window)(a0)
WindowToBack(window)(a0)
WindowToFront(window)(a0)
WindowLimits(window,widthMin,heightMin,widthMax,heightMax)(a0,d0/d1/d2/d3)
start of next generation of names
SetPrefs(preferences,size,inform)(a0,d0/d1)
start of next next generation of names
IntuiTextLength(iText)(a0)
WBenchToBack()()
WBenchToFront()()
start of next next next generation of names
AutoRequest(window,body,posText,negText,pFlag,nFlag,width,height)(a0/a1/a2/a3,d0/d1/d2/d3)
BeginRefresh(window)(a0)
BuildSysRequest(window,body,posText,negText,flags,width,height)(a0/a1/a2/a3,d0/d1/d2)
EndRefresh(window,complete)(a0,d0)
FreeSysRequest(window)(a0)
MakeScreen(screen)(a0)
RemakeDisplay()()
RethinkDisplay()()
start of next next next next generation of names
AllocRemember(rememberKey,size,flags)(a0,d0/d1)
Public function AlohaWorkbench() is intentionally not documented
AlohaWorkbench(wbport)(a0)
FreeRemember(rememberKey,reallyForget)(a0,d0)

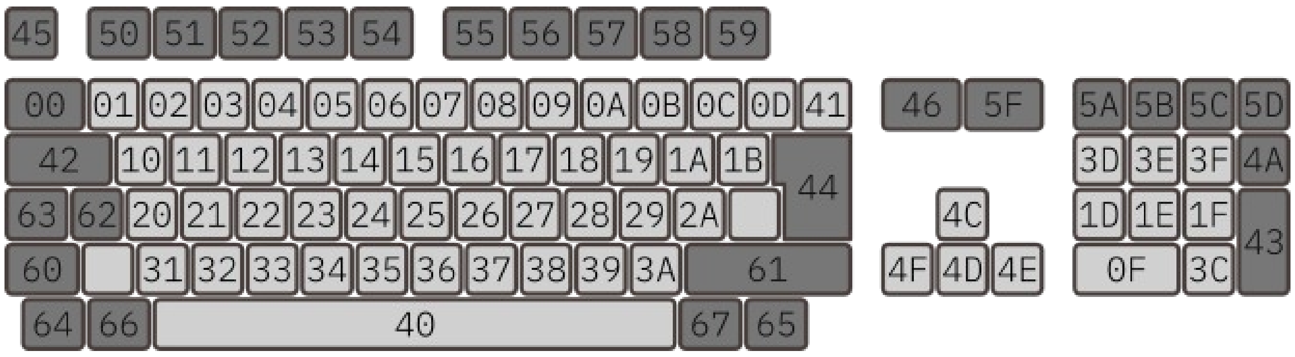
start of 15 Nov 85 names
LockIBase(dontknow)(d0)
UnlockIBase(ibLock)(a0)
functions in v33 or higher (distributed as Release 1.2)
GetScreenData(buffer,size,type,screen)(a0,d0/d1/a1)
RefreshGList(gadgets,window,requester,numGad)(a0/a1/a2,d0)
AddGList(window,gadget,position,numGad,requester)(a0/a1,d0/d1/a2)
RemoveGList(remPtr,gadget,numGad)(a0/a1,d0)
ActivateWindow(window)(a0)
RefreshWindowFrame(window)(a0)
ActivateGadget(gadgets,window,requester)(a0/a1/a2)
NewModifyProp(gadget,window,requester,flags,horizPot,vertPot,horizBody,vertBody,numGad)(a0/a1/a2,d0/d1/d2/d3/d4/d5)
functions in V36 or higher (distributed as Release 2.0)
QueryOverscan(displayID,rect,oScanType)(a0/a1,d0)
MoveWindowInFrontOf(window,behindWindow)(a0/a1)
ChangeWindowBox(window,left,top,width,height)(a0,d0/d1/d2/d3)
SetEditHook(hook)(a0)
SetMouseQueue(window,queueLength)(a0,d0)
ZipWindow(window)(a0)
public screens
LockPubScreen(name)(a0)
UnlockPubScreen(name,screen)(a0/a1)
LockPubScreenList()()
UnlockPubScreenList()()
NextPubScreen(screen,namebuf)(a0/a1)
SetDefaultPubScreen(name)(a0)
SetPubScreenModes(modes)(d0)
PubScreenStatus(screen,statusFlags)(a0,d0)
ObtainGIRPort(gInfo)(a0)
ReleaseGIRPort(rp)(a0)
GadgetMouse(gadget,gInfo,mousePoint)(a0/a1/a2)
GetDefaultPubScreen(nameBuffer)(a0)
EasyRequestArgs(window,easyStruct,idcmpPtr,args)(a0/a1/a2/a3)
BuildEasyRequestArgs(window,easyStruct,idcmp,args)(a0/a1,d0/a3)
SysReqHandler(window,idcmpPtr,waitInput)(a0/a1,d0)
OpenWindowTagList(newWindow,tagList)(a0/a1)
OpenScreenTagList(newScreen,tagList)(a0/a1)
new image functions
DrawImageState(rp,image,leftOffset,topOffset,state,drawInfo)(a0/a1,d0/d1/d2/a2)
PointInImage(point,image)(d0/a0)
EraseImage(rp,image,leftOffset,topOffset)(a0/a1,d0/d1)
NewObjectA(classPtr,classID,tagList)(a0/a1/a2)
DisposeObject(object)(a0)
SetAttrsA(object,tagList)(a0/a1)
GetAttr(attrID,object,storagePtr)(d0/a0/a1)
special set attribute call for gadgets
SetGadgetAttrsA(gadget,window,requester,tagList)(a0/a1/a2/a3)

for class implementors only
NextObject(objectPtrPtr)(a0)
MakeClass(classID,superClassID,superClassPtr,instanceSize,flags)(a0/a1/a2,d0/d1)
AddClass(classPtr)(a0)
GetScreenDrawInfo(screen)(a0)
FreeScreenDrawInfo(screen,drawInfo)(a0/a1)
ResetMenuStrip(window,menu)(a0/a1)
RemoveClass(classPtr)(a0)
FreeClass(classPtr)(a0)

DISKFONT

OpenDiskFont(textAttr)(a0)
AvailFonts(buffer,bufBytes,flags)(a0,d0/d1)
functions in v34 or higher (distributed as Release 1.3)
NewFontContents(fontsLock,fontName)(a0/a1)
DisposeFontContents(fontContentsHeader)(a1)
functions in v36 or higher (distributed as Release 2.0)
NewScaledDiskFont(sourceFont,destTextAttr)(a0/a1)

APPENDIX 3: RAWKEY CODES


45
50 51 52 53 54
55 56 57 58 59
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 41
46 5F
5A 5B 5C 5D
42 10 11 12 13 14 15 16 17 18 19 1A 1B
44
3D 3E 3F 4A
63 62 20 21 22 23 24 25 26 27 28 29 2A
4C
1D 1E 1F
43
60 31 32 33 34 35 36 37 38 39 3A 61
4F 4D 4E
0F 3C
64 66 40 67 65

FINAL WORDS

Software used for editing: LibreOffice Writer

Software used to create the line vector art: Inkscape

Software used to create the screenshots: WinUAE via Wine, Screenshot