
blitz
basic
2.1
ACID SOFTWARE

WELCOME

Thank you for purchasing Blitz Basic 2.1 We at Acid Software hope that it provides you with an environment, that gives you the total freedom you need to explore your ideas on the Amiga computer.

Blitz Basic 2.1 gives you the power to make commercial quality games such as Super Skidmarks or BlitzBombers and the flexibility to create applications such as paint packages and spreadsheets.

WHAT YOU SHOULD HAVE RECEIVED

PROGRAM DISK

This disk is installed with a limited Workbench 1.3 environment so that floppy disk users can boot straight from this disk. It contains six main files:

Acidlibs	-	The standard Blitz2 commands.
Blitz2	-	The compiler.
Deflibs	-	The additional commands created from third party libraries.
Help	-	The on-line help program.
Help.dat	-	The data file for the on-line help program.
Ted	-	The editor.

EXAMPLES DISK

This disk is full of demos and examples written in BlitzBasic 2.1 They provide a wealth of information for Blitz Basic 2.1 programmers to advance their skills. The examples have been placed into seven separate directories:

Amigamode	-	Applications related examples on gadgets, windows, arexx, etc.
AndrewsDemos	-	Games/graphics related examples by Andrew Blackbourne.
Blitzmode	-	Game/graphics related examples on scrolling, blitting, animation, etc.
MarksDemos	-	Applications related examples by Mark Sibly.
SimonsDemos	-	Games related examples by Simon Armstrong.
TedsDemos	-	Demo/graphics related examples by Ted Bailey.
Tools	-	Application and games related tools and source code.

EXTRAS DISK

This disk is full of additional utilities and programs for users with more than 1Mb of memory and a hard drive. To squeeze as much as possible onto the disk, the files have been compressed using an `archiver` called Lha. The evaluation version of this program is required so that you can unpack the files. This disk contains five archives:

BigDeflibs.lha	-	All the amigalibs and third party libraries compiled into one file.
Blitzlibs.lha	-	All the object code for amigalibs and the third party libraries and the Blitz resident files and utilities.
LibsDev.lha	-	An archive for advanced users wanting to create their own libraries. It contains documentation, example library source code and test example code for the third party libraries.
NewDebugger.lha	-	An advanced debugging utility that allows you to view copperlists, memory etc.
NewTeditor.lha	-	An Amiga style guide compliant editor that requires Workbench 2.0 or greater.

MANUAL

This manual contains detailed descriptions of all the standard commands found in Acidlibs. It also contains some helpful examples and hints on how to get the best performance from Blitz.

REGISTRATION CARD

Please fill out this card and mail it back to your Acid Software Distribution Centre. Once we have received your card, you will be allocated a Blitz User number, please quote it in all subsequent communications. You will not be eligible for any support, bug fixes and updates without registering yourself as a Blitz Basic 2.1 user.

INSTALLING BLITZ

NB: Please make backup copies of all your disks before using Blitz.

Floppy disk users

You can start using Blitz Basic 2.1 straight away. Insert the program disk and switch on your Amiga. Double-click on the Blitz2 icon and after a short pause an `okee dokee` requester will appear. Click on this and you are now ready to start. Please refer to chapter one `Getting Started` for instructions on how to load the examples.

Hard disk users

1. Create a new directory on your hard disk. Blitz is a good name to choose.

2. Copy the following files from your Blitz Program floppy disk into the new directory on your hard disk:

   acidlibs
   Blitz2
   Blitz2.info
   deflibs
   help
   help.dat
   Ted
   Ted.info

3. Create two new sub-directories in your blitz directory:

   Blitzlibs
   Userlibs

4. Add the following two assigns to your start-up sequence:

   Assign Blitz2: <hard drive>:<new directory>
   Assign Blitzlibs: <hard drive>:<new directory>/<new sub-directory>

e.g: Assign Blitz2: DH1:Blitz
Assign Blitzlibs: DH1:Blitz/Blitzlibs

5. Restart your Amiga and you are ready to run Blitz from your hard disk

INSTALLING THE ADDITIONAL UTILITIES AND EXTRAS

You do not need to install the additional utilities and extras in order to use Blitz. You can simply boot from your Blitz Program disk and start writing your programs. The extras disk is provided for users with more memory and hard disk drives and for the more advanced programmers out there.

All the additional files will need to be un-archived before they can be installed or used. In order to do this you will need the evaluation version of Lha by Stefan Boberg, which unfortunately wouldn't fit on the disk. It is available from most Public Domain software suppliers, Aminet and most magazine coverdisks.

One way to extract the files from the archives once you have the Lha program, is:

1. Load up Workbench.

2. Copy the Lha program file into your Workbench:C drawer.

3. Open a shell by double-clicking on the shell icon in the system drawer.

4. Type: Lha x
   e.g: Lha x Extras:BigDeflibs.lha Ram:

5. Copy the un-archived file into its correct place
   e.g: copy deflibs DH1:Blitz

What's in the archives and where do the files go?

BigDeflibs.lha:	This contains a single 159K deflibs file that is a direct replacement for the small 55K deflibs file in your Blitz drawer. Floppy disk users do not have enough room on their program disk to install this file.
Blitzlibs.lha:	This contains two directories and several resident files and utilities. Floppy disk users should format a blank disk and rename it Blitzlibs and then un-archive the file onto this disk. Hard disk users should un-archive this file into the Blitzlibs drawer they created when installing Blitz.
LibsDev.lha:	This contains six directories and a documentation file. Floppy disk users should format a blank disk and un-archive the file onto this disk. Hard disk users should create a new sub-directory in their Blitz drawer (a good name would be developers) and then un-archive this file into the new directory.
NewDebugger.lha:	This contains the new 152K defaultdbug file, documentation on its use and an example directory containing test programs. The file is a direct replacement for the small 33K defaultdbug in your Blitz drawer. The examples and documentation can be installed in any suitable directory. Floppy disk users will not be able to install this file as there is not enough room on their program disk. NB: The big 159K deflibs file must be installed to use the advanced debugging utility.
NewEditor.lha:	This contains the new 57K Ted file and some documentation on its use. The file is a direct replacement for the 59K Ted file in your Blitz drawer. The documentation can be installed in any suitable directory. Floppy disk users will not be able to install this file and the additional libraries it requires as there is not enough room on their program disk. NB: The new editor requires Workbench 2.0 or greater.

Directory Tree for Hard Disk users

```
DH1:—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 tool source code
     |         |— userlibdocs        docs for third party libs
     |         |— userlibprogs       test programs for third party libs
     |         └— userlibsource      source code for third party libs
     |
     |— Examples
     |         |
     |         |— amigamode
     |         |— andrewsdemos
     |         |— blitzmode
     |         |— marksdemos         example code
     |         |— simonsdemos
     |         |— tedsdemos
     |         └— tools
     |
     └— Userlibs
```

NB: The developers and examples directories are optional. Blitzlibs is mainly required for the resident files that some programs use. Userlibs is only required for testing beta-versions of your own libraries.

BUM SUBSCRIPTIONS AND SUPPORT

If you want to receive the latest additions, fixes and example code, you can subscribe to the Blitz User Magazine currently at issue eight. We are hoping to have BUM9 ready shortly after Christmas 1995 and it will consist of two disks that will unpack onto four disks (this is the standard BUM format). For £10.00, registered users can have the next two issues of BUM delivered to their door. Please note, existing BUM subscribers have already paid for the next two issues.

UK and European registered users should contact:

Acid Software
c/o Guildhall Leisure Services
Unit 15, Guildhall Industrial Estate
Kirk Sandall
Doncaster
DN3 1QR

Phone: +44 (0)1302 890000
Fax: +44 (0)1302 890010
Email: Ted@blitzuk.demon.co.uk

US and Australasian registered users should contact:

Email: acid@iconz.co.nz

PROBLEMS YOU MAY ENCOUNTER

Error Type	Explanation and Solution
Can't compile the program, there are ??????'s instead of Blitz commands	The program may be using a Blitz command from the third party libraries. you got the large deflibs installed in your Blitz drawer ? NB: Floppy users cannot install the large deflibs file as there isn't enough room on their program disk
Can't load resident	A few programs use resident files which are contained in the blitzlibs archive. Floppy users have you named a blank disk Blitzlibs and un-archived the file blitzlibs.lha onto it? Hard Disk users have you un-archived the blitzlibs.lha file into the correct drawer on your hard drive ? Have you added the assign for Blitzlibs: to your startup sequence ?
Please insert volume Blitzlibs:	See above explanation and is your deflibs file in the same drawer as the Blitz2 program file ?.
Can't load graphic/shape/sound/include file	Blitz can't find the file to be loaded. Have you used the Change Directory gadget on the file requester ?

CONTENTS

1. GETTING STARTED

Installing Blitz

There are two install programs included on Disk1 of the Blitz disks, HDInstall and FloppyInstall. From Workbench click on the one that is applicable. Those installing onto harddisk will want to make sure they have at least 4 megs free in the partition they are installing to while floppy users will be informed by the FloppyInstall program of the number of blank disks they will need to unpack all the data onto.

Registration Card

Please fill out this card and mail it back to Acid Software. You will not be eligible for any support, bug fixes and updates without registering yourself as a Blitz2 user. We would also like to here about what you want to use Blitz2 for and any things you think need adding to Blitz2.

OkeeDokee?

Once you have a working backup of the disks or have installed Blitz 2 onto your harddisk its time to take your new programming language for a spin.

Double click the Blitz2 icon to run the editor/compiler. The editor screen should appear with a copyright notice (which should NOT be ignored!). Click on OkeeDokee and you're up and running.

Running the Examples

Okee dokee, if you have got this far without any problems you're ready to drive the speed machine (thats the Blitz 2 editor/compiler we're talking about).

Select the LOAD menu item, insert the examples disk or select the Blitz2:Examples drawer on your harddisk and load in one of the examples. Any file ending with the suffix .bb2 is a source file able to be loaded into the Blitz2 editor/compiler. Once you have loaded a .bb2 file have a read, guess what it's going to do then select compile and run from the compiler menu.

The only problem you should have with running the examples is to do with 'Current Directories'.

Current Directories

If you changed the pathname in the file-requester to locate the example you loaded, you will usually need to click on the CD gadget before selecting OK on the file-requester. This changes the current directory to that which you loaded the example.

By changing the current directory, any files the program attempts to use will be loaded from the same directory as that from which the source code was loaded.

This means that when the program is executed (run) and attempts to load any data or graphics from disk, the default path (directory it looks for files) will be set to the same directory as where the program itself was loaded.

If the example comes up with a runtime error "Couldn't Load Shape" or the like it will be because the current directory has not been to set. Hit Escape to exit the debugger and return to the editor.

If the example crashes the machine, it is because the same error occurred but error checking was disabled on the compiler options menu.

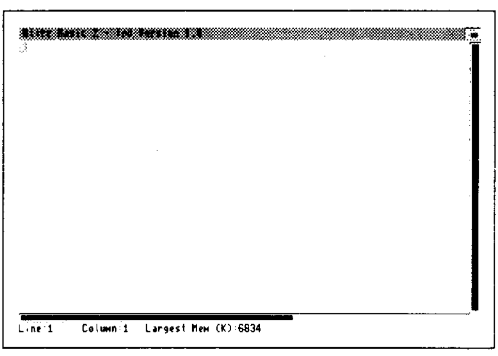
Using Ted the Blitz2 Editor

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 standalone 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 like 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.



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 letters will appear when 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 letters 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 cover up 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 right by so many columns.

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 AmigaJ 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 important keys used with the Blitz2 editor.

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 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. The following is a list of the features accessible from these menus in order from left to right.

The PROJECT Menu

NEW	Kills the file you are 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.
ABOUT	Displays version number and credits concerning Blitz2.
PRINT	Sends your file to an ouput device usually PRT: the printer device.
CLI	Launches a command line interface from the editor, use the ENDCLI command to close this CLI and return to the Blitz2 editor.
CLOSEWB	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 ChipMem.
QUIT	Close the Blitz2 editor and returns you to workbench or CLI.

The EDIT Menu

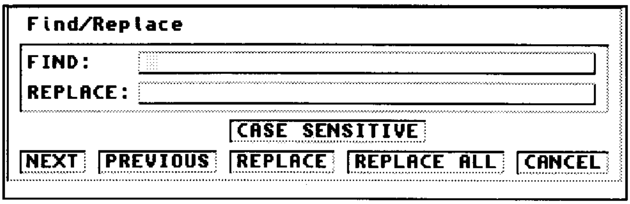
COPY	Copies a block of text that is highlighted with the mouse or f1-f2 key combination to the current cursor position. The F4 key is another keyboard shortcut for COPY.
KILL	Deletes a highlighted block of text (same as shift F3 key).
BLOCKTODISK	Saves a highlighted block of text to disk in ascii format.
INSERTFROMDISK	Loads a file from disk and inserts it into the file you are editing at the current cursor position.
FORGET	De-selects a block of text that is selected (highlighted).
INSERTLINE	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.

FIND	Will search the file for a string of characters.
NEXT	Positions the curosr at the next occurence of the Find-String entered using the FIND menu option (as below).
PREVIOUS	Will position the curosr at the last occurence 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 occurence 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.

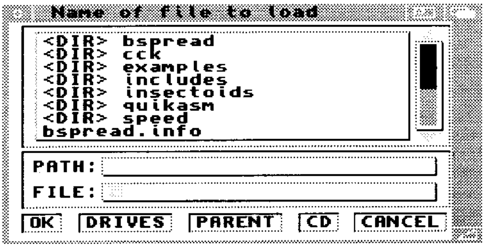
The CASE SENSITIVE option will only find strings that have the same lettters capitalised, default is that the search will ignore whether letters are caps or not.

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 occurence of the Find: string, delete it, and insert the Replace: string in it's place.

REPLACE ALL will carry on through the file doing replaces on all occurences of the Find: string.

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.



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 a <DIR> will change to that directory and list the files contained in it.

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. This means that when you select CLI or launch a task from the editor its root directory will be that selected by the CD gadget.

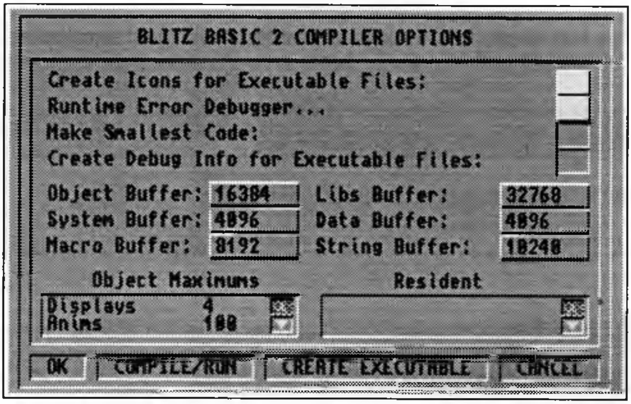
The last feature of the Blitz2 FileRequester is the ability to size its window, by dragging the bottom right of the window with the left mouse button you can see many more files at one time.

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.
CREATERESIDENT	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 attained.
VIEW TYPE	Allows you to view all currently resident types. Click onthe 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 the Blitz2 compiler environment. This is useful when writing your own Blitz2 libraries 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.



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 accessable from the WorkBench. Note: for the program to execute correctly when run from workbench 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 MakeSmallest (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 precompiled resident files to the Blitz2 environment. Click in the box and type in the resident file name.

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 avaiiable 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:

A **A** SELECTs all text that is indented the same amount as the current line (strictly for structured programming housekeeping)

A **B** BOTTOM will position cursor on last line of file

A **D** DELETE LINE removes the line of text on which the cursor is currently positioned

A **F** FIND/REPLACE executes the FIND command in the SEARCH menu

A **G** GOTO LINE moves cursor to specific line of file

A **I** INSERT LINE moves all text at and below the cursor down one line

A **J** JOIN LINE adjoins next line with current line

A **L** LOAD reads a file from disk

A **N** NEXT searches for the next occurence of the 'find string'

A **P** PREVIOUS searched for previous occurence of the 'find string'

A **Q** QUIT will exit the Blitz2 editor

A **R** REPLACE will replace text at cursor (if same as find string) with the alternate string specified with the Find command.

A **S** SAVE writes a file to disk

A **T** TOP moves the cursor to the top of the file

A **W** FORGET will unhighlight a selected block of text

A **Y** DELETE TO RIGHT of cursor

A **Z** CLI

A **?** DEFAULTS allows the user to change the look and feel of the Blitz2 editor

A **]** BLOCK TAB moves whole block right one tab

A **[** BLOCK UNTAB moves whole block left one tab

2. BLITZ BASIC'S

My First Program

Type in the following two lines:

```
Print "This is my first program written in Blitz2!"
MouseWait
End
```

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. Thats 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 calulation. 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 on 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 want to print a list of numbers, in a column.

The **NPrint** command 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 it's 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 it's right:

```
a=NOT 5
```

or from the variables on it's 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 precedense 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.

<table>
<tr><td>NOT
-</td><td>RHS logically NOTted
RHS arithmetically negated</td></tr>
<tr><td>BITSET
BITCLR
BITCHG
BITTST</td><td>LHS with RHS bit set
LHS with RHS bit cleared
LHS with RHS bit changed
true if LHS bit of RHS is set</td></tr>
<tr><td>^</td><td>LHS to the power of RHS</td></tr>
<tr><td>LSL
ASL
LSR
ASR</td><td>LHS logically shifted left RHS times
LHS arithmetically shifted left RHS times
LHS logically shifted right RHS times
LHS arithmetically shifted right RHS times</td></tr>
<tr><td>&
|</td><td>LHS logically ANDed with RHS
LHS logically ORed with RHS</td></tr>
<tr><td>*
/</td><td>LHS multiplied by RHS
LHS divided by RHS</td></tr>
<tr><td>+
-</td><td>LHS added to RHS
RHS subtracted from LHS</td></tr>
<tr><td>=
<>
<
>
<=
>=</td><td>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</td></tr>
<tr><td>AND
OR</td><td>LHS logically ANDed with RHS
LHS logically ORed with RHS</td></tr>
</table>

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 the 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. The OR operator will return true if either the left or the right hand side is true. The NOT operator returns false if the following operand is true and true if the operand is false.

Binary Operators

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

The binary system means that all numbers are represented by a series of 1s and 0s. A byte is made up of 8 such bits, a word 16 and a long word 32.

Further discussion of the binary operators in Blitz2 can be found in any text covering the 68000 microprocessor.

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 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, 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
End
```

Note how the 2 lines inside the loop are indented across the page. This practise 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 it's 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 the 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 for ever 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 LeftAlt 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 not numbers are called string variables. String variables require the $ signs after their names. The following shows a simple example of a string variable:

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

The variable c$ will now contain the string "SimonArmstrong". 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 "GoodBye"
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
```

Because the **Goto** statement makes the program jump to the label start, "Hi There" is never printed.

The **Gosub** command is used to jump to a subroutine, a subroutine is a piece of code terminated with a **Return** statement. This means that after executing the subroutine, the 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 the labels are preceeded with a period. This makes them *mousable* labels which 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 finding your way around 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 what they type into a string variable:

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

The number 80 in Edit$(80) refers to the 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, 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 as *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     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. 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 Fast Floating Point libraries.

A variable is assigned a certain type by adding the relevant suffix to it's 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.

The 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 d.p. accuracy
light_speed.f=3e8     ;e is exponent i.e. 3x10^8
```

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 following section).

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

The Data Statement

The **Data** statement is 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 the 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 it's 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 floating point but faster. The Float type is the Floating Point type supported by the Amiga Fast Floating Point libraries.

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

Using the 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 necesary 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="Harry",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, they must however be separated by colons. The following is another example of a NewType definition:

```
NewType .nme
    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 the $ or .s suffix as normal string variables do, including the suffix will cause a *Garbage at End of Line* 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 [&] 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 addess.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 ommitted 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, pathnames 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 pathnames can be avoided.

When a backslash *preceeds* 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
```

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

Note that UsePath is a *compiler directive*, this means 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 may be any number of dimensions.

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

As with all variable definitions an array will be of default type unless a *.type* suffix is added to the array name:

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

The ability to use arrays of NewTypes often reduces the number of arrays a BASIC program 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 are 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 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 would still need to check if the list was full each time we added an alien:

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

Note that lists can be dimensioned to hold any type not just aliens! (They're not just for games that is.)

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

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 thay 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, 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 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

The pointer type in Blitz2 is a complex beast. 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.

As an example we may have a large list of customers, our routine goes through them one by one, 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 **Print** *biggest\name.

4. PROCEDURES

Introduction

A procedure is a way of 'packaging' routines into self contained modules.

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 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
Statements

A procedure that does not return a value is called a Statement in Blitz2.

Here is an example of a statement type procedure which prints out the factorial of a number:

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

For k=1 To 5
    fact{k}
Next
MouseWait
```

Note the use of curly brackets { and } to both define parameters for the procedure, and in calling the procedure. These are necessary 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 (see Shared below).

Also, variables used to pass parameters may only be of primitve 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 To 5
    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 the Function command. If this is omitted, the 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 it's 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 inital 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.

The following code is an example of an assembler procedure in Blitz:

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, runtime errors occur when your 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 2 functions (commands that return a value) must have their parameters inside brackets:

   If ReadFile(0,"ram:test")

2. Blitz2 commands that are not functions must not have their parameters in brackets:

   BitMap 0,320,256,3

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

   person\name$="Harry"   ;*(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 syntax of a command and maybe cross reference your code with one of the examples.

Don't forget the Help key 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 Blitz2, the Runtime Error Debugger should always be enabled on the Compiler Options window. If it is 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 filehandling should **always** include some sort of error trapping to handle situations where the program cannot locate a file, or the file is 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 of the section.

The following will flash the screen and end if **LoadShapes** fails:

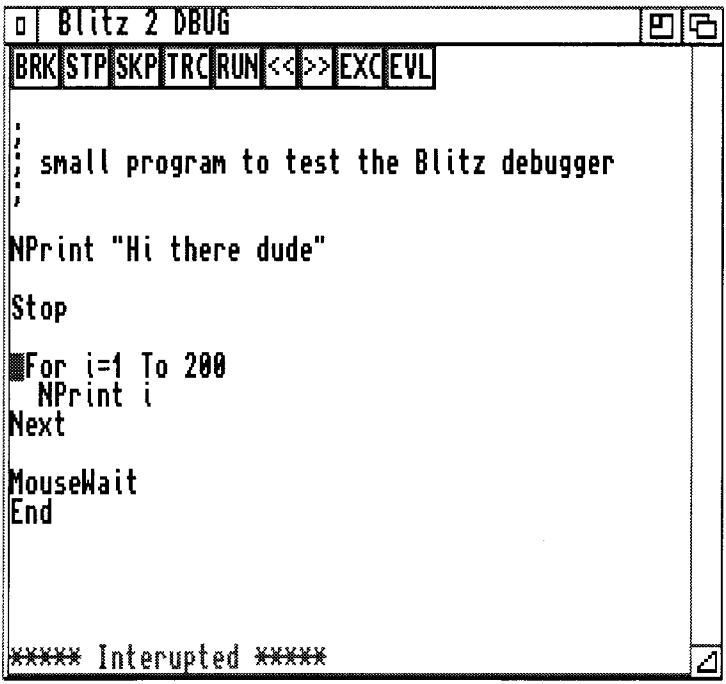
```
SetErr
    DisplayBeep_ 0
    End
End SetErr

LoadShapes 0,"filename"

ClrErr
```

If a runtime error occurs when a program is run from the editor the Blitz2 debugger will be activated. Of course RuntimeErrors 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 keyboard combination, 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.



Note that 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 when then RUN.

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 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 it's name after clicking on EVL.

Tracing program execution

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

Step 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 use the STP gadget.

Level is used to change the trace level, if Level is toggled on, the debugger will not trace or single step through the inside of For..Next loops but execute normally until the 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 can return normally after the debugger is activated using the Run gadget.

If the debugger was activated using the STOP command the arrow will be pointing to the STOP, before continuing the command must be skipped over using the Igonore 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.

The 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 the direct mode commands Evaluate and eXecute:

Can't Create in Direct Mode

Occurs if you try and Evaluate a variable that does not exist (hasn't been created) in the program.

Library Not Available in Direct Mode

Occurs when a Blitz2 command is eXecuted and is from a command library not used by the program. If the program does not use strings for instance, 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.

Not Enough Room in Direct Mode Buffer

This error should never occur, if it does the object buffer 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 (see viewing command history).

6. BLITZ OBJECTS

Blitz2 Objects Overview

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 Inuition 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 the memory later in the program, otherwise it is usual to let Blitz2 free up all objects automatically when the 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 the 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 Pallettes are three other Blitz2 objects that often need to be currently *used*, for commands to work properly.

It should also be noted that when an object is created, it also becomes the currently used object of it's class.

Blitz2 makes extensive use of this *currently used* object idea. It's advantages include faster program execution, less complex looking commands, and greater program modularity.

Input/Ouput Objects

BitMap, File and Window objects can all operate as I/O devices. The ***Object*Input** and ***Object*Output** 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 of a particular objects structure.

Advanced users can use the **Addr** command with **peek** and **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 Inuition Screen structure from the Blitz2 Screen object in memory.

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

This next listing illustrates obtaining a Window's system structure and assigning it to a pointer type *.Window*. **AmigaLibs.Res** should be resident before running 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.

Screens

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

Free Screen *n* should only be attempted after any Windows open on the Screen are closed (freed) 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

Windows are 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 the **WindowInput** and **WindowOuput** commands. The cursor position for input/ouput can be controlled with the **WLocate** command.

Windows can be freed without the 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.

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 command, you can not free bitmaps created with the **ScreensBitMap** command.

As with windows, bitmaps can be used as input/ output devices with the **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 the **GetAShape** command.

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

There are many powerful commands in Blitz2 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 the **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 used* slice. The **RGB** command also affects the colour registers in the *currently used* slice as does the **Use Palette** command.

Files

Unlike other Blitz2 objects files are opened and closed rather than initialised and killed.

Files are initialised with the **OpenFile()**, **ReadFile()** and **WriteFile()** functions. Unlike other Blitz2 objects a function is used so the program can tell if the file was successfully opened.

The **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 by 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 Blitz2 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

Introduction to Blitz Mode

Although the Amiga's operating system is very powerful, it's 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 permanant 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 payed regarding entering Blitz mode, 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

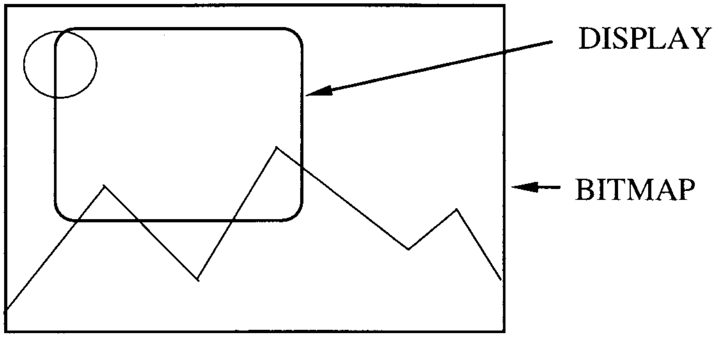
The designers of the Amiga hardware have implemented many features for acheiving smooth, fast graphics. After entering Blitz mode the display is controlled using Slices. Slices are much more flexible than the operating system's Screens, they allow features such as smooth scrolling, double buffered displays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

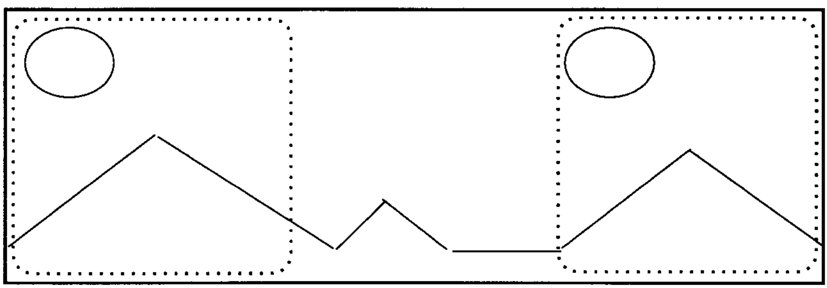
Smooth scrolling is acheived by displaying only a portion of a large bitmap. The Amiga hardware enables us to move the display window around the inside of a large bitmap as the following diagram shows:



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, and 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.

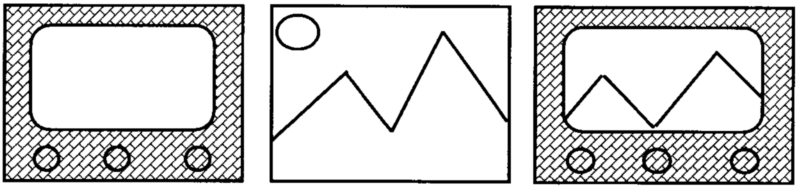


The above left right scenario also applys to vertical scrolling (up and down).

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 acheive this effect.

In a dual-playfield display, two 8 colour bitmaps can be displayed, one infront of the other, 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.



BMAP0 **BMAP1** **BMAP0onBMAP1**

Copper Control

Smooth animation is acheived 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 is useful to sync things to the vertical position of the vertical beam. This is acheived 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 acheived using the **ColSplit** command.

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

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 using the blitter.

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

Blit used to put shapes onto bitmaps.

QBlit same as Blit but Blitz2 remembers where the shape was put and will erase it when it is 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 bimap.

Scroll used to copy sections of a bitmap from one position to another.

QAmiga Mode

It is also possible to jump out of Blitz mode and back into Amiga mode. This can 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 your Blitz mode programs to jump into Amiga mode for operations such as file I/O, then to jump back to Blitz mode without having to destroy a 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 launces straight into Blitz mode.

• A590 Hard drive users - always wait for the second blink of the drive light when using Workbench 1.3, 2.0 users have there buffers flushed in one go.

• If you use the QAmiga statement for the purpose of **writing** data to disk, it's a good idea to execute a delay before going back to Blitz mode - In effect, simulating the above. Executing a VWait 250 will provide a delay of about five 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 operating system become unavailable while you're in Blitz mode. For example, 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 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

#const=10
```

Now, 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.

The resident file AMIGALIBS.RES contains all the structures, constants and macros associated with the Amiga operating system. Those familiar with programming the operating system will find not havine to include all the usual library header files 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 Operating System 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 (see appendix 5 in the Blitz2 Reference Manual).

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

The following is an example of calling routines in the Amiga ROM's graphics and 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 acheiving this objective.

Accessing Operating System structures

With the file AMIGALIBS.RES resident (see start of chapter) even more control of the operating system is possible. The following is an example of accessing operating system 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 *mynode\ln_Succ
      a$=Peek$(*mynode\ln_Name);print node name
      NPrint a$
      *mynode=*mynode\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$** is an excellent way of retrieving text from operating system structures, it reads memory directly into a Blitz2 string variable until it hits a null (**chr**$(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 similiar to the VarPtr function supplied in other BASIC's.

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 adress-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. #width=320 means the variable #width is a constant and will always be equal to 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 numbers '320', the programmer can just change the constant equate at the top of the program #width=320 to #width=640 etc.

Conditional Compiling

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 two differnt 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 unecesary parts of the code it becomes easier to pinpoint where exactly the error is occurring. However we hope the Blitz2 debugger will make this practise 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 is important to remember that the 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 saveroutine in the object code so that there is no way it can be un-crippled by hackers.

The conditional compiler directives however come into their own when doing macro programming.

Macros

Macros are a feature usually only found in Assemblers or lower level programming languages. They can be 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 the squigly brackets { 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 the TAB key on the Amiga keyboard) 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 any text, 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 '0 Parameter

The '0 parameter is special, it returns the number of parameters passed to the 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 macro checks to see if two parameters were passed and generates a compile time error is there was not:

```
Macro Vadd
CNIF '0=2
  '1='1+'2
CELSE
  CERR "Illegal number of '!Vadd' Parameters"
CEND
End Macro

!Vadd{a}
```

If you compile and 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) is 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 do not 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*6-`3*7:End Macro

#xm=320:#ym=256

Screen 0,28:ScreensBitMap 0,0

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

MouseWait
```

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.

```
    var2=20
    var3=30

    Macro lvar
        NPrint var~`1~
    End Macro

    !lvar{2+1}

    MouseWait
```

The above example *without* the cmake characters ~ would print the value 21 as Blitz2 would expand the the code after the NPrint to read var2+1, instead it evaluates the expression between the cmake characters and generates a 3 which it then inserts into the macro text.

It is 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:

- in line using the GetReg and PutReg commands to access variables
- inside statements and functions
- developing custom Blitz2 libraries

GetReg & PutReg

The **GetReg** and **PutReg** commands allow the assembly programmer access to the BASIC variables in the program. The following listing illustrates the use of GetReg and PutReg:

```
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 the 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.l#2,d1
    SUBQ#1,d1
    MOVE.l 8(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.

Because address register a4 is used as the local variable base, the UNLK a4 command must be at the top of a procedure, the procedure must be 100% assembler code and the address registers a4-a6 must not be destroyed.

The following listing illustrates the use of assembler in a 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 8(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
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 programmers pack from Acid Software. Blitz2 contains an extremely powerful library system giving the experienced machine code programmer a highly productive and powerful environment to develop advanced software.

9. PROGRAMMING TECHNIQUE & OPTIMIZING

Label and Variable Names

The following are rules that must be conformed 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.

Consistancy 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", inital 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 "loop1", "loop2" etc.

Nomenclature is a personal thing, by sticking to a certain style with variable and label names many problems associated with debugging 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

Unlike other BASIC's that use the REM statement, Blitz2 uses the semicolon character. Any text after a semicolon on a line will be ignored by the Blitz2 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, 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

The 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 several features for indenting code. The tab key is used to move the cursor across the page. By changing the tab setting in Ted's defaults requester the size of indent can be altered.

By highlighting a block of code, block tab and untab (*A*[and *A*]) 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 are bug hunting or making small adjustments to a certain section 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 i.e. 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

Besides keeping routines well documented it is always a good idea to keep a piece of paper handy to jot down the important bits. Lists of variables that are common between routines as well as things still 'to do' in unfinished routines should always be written down.

The 'to do' list is always a good way of thinking out all the 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 to start a routine that needs all sorts of other routines to function. By starting with the standalone/ 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 one of them. Developing a modular approach to programming is definately the most effective way of finishing a piece of software.
Keeping your code readable

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

Optimising Code

It is 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 the user wait will detract from the productivity of the package in general.

Algorithms

The most important key to optimising different routines is the overall approach taken to implementing them in the 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.). The time it takes to perform the code inside a loop is multiplied by the number of times it loops. This may seem rather logical but often programmers will equate the number of lines of code in a routine to the time taken to execute it.

The code:

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

Will take exactly the same amount of time as typing:

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

one hundred times, which will equate to 300 lines of code!

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!

Lookup 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 LUPs 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*180/pi) with x=sinlup(a) which will of course be more than 10 times as quick. The array would be setup in the program initialisation as follows:

```
Dim sinlup(360)
For i=0 To 360
    sinlup(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 is important to test two 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 by hand.

When writing arcade games that will be performing a main loop each frame, it is useful to poke the 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 the 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 codeas 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 $0f0,$dff180  ;turn background green
```

- Use QBlits if possible as they are the fastest way of implementing animated graphics in Blitz2.

- If aliens change direction using complex routines, split up the aliens into groups and every frame select a different group to have their directions changed, the others can move in the same direction until it is their turn. This method applies to any routines that do not 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 the processor and blitter. A smaller display increases the amount of time given to the processor and blitter.

There is an infinite number of ways to increase the speed of Blitz2 code, subscribing to Blitz User Magazine is one of the best ways of speeding up your code!

Those developing games on machines with fast mem and faster processors should remember that most people do not have either! It is a good idea to disable fastmem when testing the speed of your code.

10. PROGRAM EXAMPLES

Number Guessing

The following is a small program where the 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 up, you'll find it pretty hard to guess the number, this is because the number Blitz2 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 indepth 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, it's 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, 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 **REMark** command in Blitz2 and does not retain any of it's older functionality in **Print** statements.

Creating a standalone Workbench program

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 prog 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 the default 0 character that appears in the window form of the **Edit**() function.

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

Now, select *Create Executable* from the compiler Menu or use the *AmigaE* keyboard shortcut.

Type the name of the program you wish to create and hit return. You have now created your first stand alone program with Blitz2, go to the Workbench and click on the 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 the screen with the **Plot**, **Line**, **Box** and **Circle** commands. Programs that use windows should not use this method, rather they should draw into specific windows using the **WPlot**, **WLine**, **WBox** and **WCircle** commands.

Using Menus and the Blitz2 File Requester

The following program opens its own screen & window, attaches a menu list, and depending on what the user selects from the menus, either opens the Blitz2 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
MenuItem 0,0,0,0,"Load ","L"
MenuItem 0,0,0,1,"Save ","S"
MenuItem 0,0,0,2,"Quit ","Q"

MaxLen path$=192   ;MUST be executed before a file requester is used
MaxLen name$=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 ;its 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 ;its item 1 which means save
                p$=FileRequest$("FileToSave",path$,name$)
                NPrint "Attempted to Save ",p$

            Case 2;its item 2 which means quit
                End

        End Select

    End Select
Forever
```

The **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.

The 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, clicks on a gadget an 'event' is sent to your program. Usually an application program will use the **WaitEvent** command, which makes the 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. A menu event returns 256 ($100 hex), a close window event returns 512 ($200 hex). A full list of events and their IDCMP codes is listed on page 25-5 of the Blitz2 reference manual.

String Gadgets

The following program demonstrates the use of string gadgets. These allow the user to enter text via the keyboard. The following sets up 3 string gadgets for decimal, hex and binary input/output.

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 the 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 the user to adjust the colors 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 color register selected.

The 32 color 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=Int(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,On:Redraw 0,3+cc

Repeat
    SetVProp 0,0,1-Red(cc)/15,1/16
    SetVProp 0,1,1-Green(cc)/15,1/16
    SetVProp 0,2,1-Blue(cc)/15,1/16
    Redraw 0,0:Redraw 0,1:Redraw 0,2
    ev.l=WaitEvent
    If ev=$40 AND GadgetHit>2
        Toggle 0,3+cc,On:Redraw 0,3+cc
        cc=GadgetHit-3
        Toggle 0,3+cc,On:Redraw 0,3+cc
    EndIf
    If (ev=$20 OR ev=$40) AND GadgetHit<3
        r.b=VPropPot(0,0)*16
        g.b=VPropPot(0,1)*16
        b.b=VPropPot(0,2)*16
        RGB cc,15-r,15-g,15-b
    EndIf
Until ev=$200
```

Database Type Aplication

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 IntuiTools 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 intuitools 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,"| <"
TextGadget 0,129,75,0,12,"<<"
TextGadget 0,161,75,0,13,">>"
TextGadget 0,193,75,0,14,">| "
TextGadget 0,226,75,0,15,"DIAL"
TextGadget 0,270,75,0,16,"LABEL"

SizeLimits 32,32,-1,-1
Window 0,0,24,331,91,$100E,"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")
      FileInput 0
      While NOT Eof(0)
            If AddItem(people())
                  For i=0 To #num-1:\info[i]=Edit$(128):Next
            EndIf
      Wend
EndIf

ResetList people()

;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

The Following program is an example of accessing Operating System 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.

You may need a pathname. AmigaLibs is found in the Resident directory of the Blitz 2 program disk.

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.

Blitz 2 now knows that exec points to an execbase structure and using the backslash charcter we can access any of the variables in this stucture 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 or what have you.

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 very useful for placing text pointed to by a C definition such as *ln_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\ln_Name)
 NPrint a$
 *mynode=*mynode\ln_Succ
Wend

MouseWait
```

Prime Number Generator

The following program generates a list of prime numbers from 2 up to a limit specified by the 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.

The While..Wend structure is used to loop through the main algorithm until the upper limit is reached. The algorithm simple take the next integer, loops through the list of the prime numbers it has already generated until either it finds a divisible number or it is too far through the list (the item in the list is greater than the square root of the 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 number 2 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 32x32 pixel shape and so need an extra 32 pixels all round the bitmap. The **Show** 0,32,32 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 accomadate these factors.

The *.makeshape* routine creates a temporary bitmap to draw a patern 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,i:Next
    GetaShape 0,0,0,32,32
    Free BitMap 1
    Return
```

Dual Playfield Slice

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

The macro *rndpt* simply inserts the code **Rnd**(640),**Rnd**(512) into the source each time it is called. For instance **Line** !rndpt,!rndpt,**Rnd**(7) is expanded internally by the compiler to read:

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

Once againg the extended form of the slice command has to be used with flags set to $fffa giving us a lores dualplayfield scrollable display.

In dualplayfield 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)*128
    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, necessary to acheive 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 acheive 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. QBlits work by doing a normal Blit and storing the position of the 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 with out them leaving "trails" behind them from their old position.

The **move** #-1,$dff180 pokes the background color to white, this allows us to see how much of the frame has been taken since the 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 a more thorough discussion of frame rates etc.

Hmm, 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,i:Next
  GetaShape 0,0,0,32,32
  Free BitMap 1
  Return
```

Smooth Scrolling

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

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

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

Note how 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. DISPLAY LIBRARY & AGA PROGRAMMING

Introduction

The Display Library is a recent addition to Blitz. Developed as a replacement to Slices it not only offers games programmers access to all of the 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, 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 the 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 colors.

The longer version has the following format:

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

The ypos is usually set to 44 the standard top of frame for a PAL display. If the 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, colors should be set to the number required. When using more than 32 colours ensure tha #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 customcops.

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
#threebitplanes=      $03
#fourbitplanes=       $04
#fivebitplanes=       $05
#sixbitplanes=        $06
#sevenbitplanes=      $07*
#eightbitplanes=      $08*

#smoothscrolling=     $10        ;set if you will be scrolling the bitmap
#dualplayfield=       $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.

Colors

The #agacolors flag must ALWAYS be set when more than 32 colours are in use or when 24 bit color 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 BlitzMode programming chapter for an explanation of hardware scrolling.

Notes:

* Always use the extended form of DisplayBitmap with smoothscrolling set, even when 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.

DualPlayfield

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 fetchmode settings. Most AGA modes will require the display to be shrunk horizontally for 8 sprites to be displayed. Currently this can only be acheived using the DisplayAdjust command, 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 colours sprited 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 color 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.

FetchMode

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 noticable with AGA Amiga's running without additional fastmem.

When using increased fetchmodes bitplanes must always be a multiple of 64 pixels wide.

Those wanting to attempt DisplayAdjusts on displays with larger fetchmodes will encounter severe difficulties in creating a proper display. We think it is actually impossible for displays to run at fetchmode 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's 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 paramater to be set.

There are two forms of custom copper commands, the first will allow the copper to affect the display every scanline while the second defines a certain line for the copper to do it's thing. These new commands include:

The following require a negative size, this denotes that so many instructions must be allocated for every scanline 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 poistive 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$

The use of these commands is illustrated by code included in the 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 color palette is created, the first 16 colors are used by the back playfield and second 16 by thr 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*
```

Note how the InitCopList command can be executed before going into Blitz mode. All the display commands are mode independent except for 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 color 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 31: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. The DisplayAdjust is required so as 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.

The SpriteMode2 tells Blitz to create 64 pixel wide sprites for each channel. With out the SpriteMode each sprite would require 4 channels, againg this is 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 postions if possible.

```
;
; smoothscrolling 16 color screen with 8 64 wide sprites
;

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

BitMap 0,640,512,4

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

InitPalette 0,48

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

InitCopList 0,$10014

DisplayAdjust 0,-2,8,0,16,0;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 COMMANDS

A computer program is made up of a sequence of commands that are executed sequentially (one after the other). Certain commands are used to interupt 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.

Standard BASIC commands to change program flow such as Goto, Gosub are standard in Blitz, unlike older BASIC's, locations are specified as program labels and not line numbers. More modern structured BASIC features such as Procedures (known as Statements and Functions), While..Wend, Repeat..Until, Select..Case and more allow a more structured approach to programming.

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

Goto *Program Label*

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

Gosub *Program Label*

Gosub 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

Return is 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 *Program Label[,Program Label...]*

On 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 Program Label will be branched to. A result of 2 will cause the second Program Label to be branched to and so on. If the result of Expression is less than one, or not enough Program Labels are supplied, program flow will continue without a branch.

End

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

Stop

The Stop command causes the Blitz Debugger to interupt program flow. Place Stop commands in your code as breakpoints when debugging, ensure runtime errors are enabled. Click on Run from the debugger to continue program flow after a Stop.

If *Expression [Then...]*

If 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

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

Else *[Statement...]*

Else 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*

The While command is 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

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

Select *Expression*

Select 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*

A Case is 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

A Default statement may appear following a series of Case statements to cause a section of program code to be executed if NONE of the Case statements were satisfied.

End Select

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 initializes 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 the For/Next loop's Next, ending the loop. If not, program flow continues on until the loop's Next is reached. At this point, the value specified in Expression3 (the '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...]]*

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

Repeat

Repeat is 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*

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

Forever

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 the 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

MouseWait simply halts program flow until the left mouse button is pushed. If the left mouse button is already held down when a MouseWait is executed, program flow will simply continue through.
MouseWait should normally be used only for program testing purposes, as MouseWait severely slows down multi-tasking.

VWait *[Frames]*

VWait will cause program flow to halt until the next vertical blank occurs. The optional Frames parameter may be used to wait for a particular number of vertical blanks.
VWait is especially useful in animation for synchronizing display changes with the rate at which the display is physically redrawn by the monitor.

Statement *Procedurename{[Parameter1[,Paramater2...]]}*

Statement 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

End Statement declares the end of a 'statement type' procedure definition. All statement type procedures must be terminated with an End Statement.

Statement Return

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...]]}***

Function 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 the 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

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***

Function Return allows 'function type' procedures to return values to their calling expressions. Function type procedures are always called from within Blitz expressions.

Shared ***Var[,Var...]***

Shared is 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***

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

Type	Cause of Interupt
0	Serial transmit buffer empty
1	Disk Block read/written
2	Software interupt
3	Cia ports interupt
4	Co-processor ('copper') interupt
5	Vetical 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 interupt

The most useful of these interupts is the vertical blank interupt. This interupt occurs every time an entire video frame has been fully displayed (about every sixtieth of a second), and is very useful for animation purposes. If a section of program code has been designated as a vertical blank interupt handler, then that section of code will be executed every sixtieth of a second.

Interupt handlers must perform their task as quickly as possible, especially in the case of vertical blank handlers which must NEVER take longer than one sixtieth of a second to execute.

Interupt handlers in Blitz must NEVER access string variables or literal strings. In Blitz mode, this is the only restriction on interupt handlers. In Amiga mode, no blitter, Intuition or file i/o commands may be executed by interupt handlers.

To set up a section of code to be used as an interupt handler, you use the SetInt command followed by the actual interupt handler code. An End SetInt should follow the interupt code. The Type parameter specifies the type of interupt, from the above table, the interupt handler should be attached to. For example, SetInt 5 should be used for vertical blank interupt code.

More than one interupt handler may be attached to a particular type of interupt.

End SetInt

End SetInt must appear after a SetInt to signify the end of a section of interupt handler code. Please refer to SetInt for more information of interupt handlers.

ClrInt *Type*

ClrInt may be used to remove any interupt handlers currently attached to the specified interupt Type. The SetInt command is used to attached interupt handlers to particular interupts.

SetErr

The SetErr command 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

End SetErr must appear following custom error handlers installed using SetErr. Plase refer to SetErr for more information on custom error handlers.

ClrErr

ClrErr may be used to remove a custom error handler set up using SetErr.

ErrFail

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 reported and transfer will be passed to direct mode.

R-2: VARIABLE HANDLING COMMANDS

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 the programmer 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. Normally, an equals sign ('=') is placed between the variable name and the expression. If the equals sign is omitted, then an opertor (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***

Exchange will 'swap' the values contained in the 2 specified variables. Exchange may only be used with 2 variables of the same type.

MaxLen ***StringVar=Expression***

MaxLen 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 2 Blitz commands require the use of MaxLen - FileRequest$ and Fields.

DEFTYPE ***.Typename [Var[,Var...]]***

DEFTYPE may be used to declare a list of variables as being of a particular type. In this case, Var parameters must be supplied.
DEFTYPE 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***

NEWTYPE is used to create a custom variable type. NEWTYPE must be followed by a list of entry names separated by colons (':') and/or newlines. NEWTYPEs are terminated using End NEWTYPE.

SizeOf ***.Typename[,Entrypath]***

SizeOf allows you to determine the amount of memory, in bytes, a particular variable type takes up. SizeOf 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...])***

Dim is used to initialize a BASIC array. Blitz supports 2 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 1 dimension.

ResetList ***Arrayname()***

ResetList is 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 item.

ClearList ***Arrayname()***

ClearList is used in conjunction with list arrays to completely 'empty' out the specified list array. List arrays are automatically emptied when they are Dimmed.

AddFirst ***(Arrayname())***

The AddFirst function allows you to insert an array list item at the beginning of an array list. AddFirst 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())***

The AddLast function 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())***

The AddItem function 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()***

KillItem is used to delete 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())***

PrevItem 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())***

NextItem will set the specified list array's 'current item' pointer to the item after the list array's old current item. This allows for 'forwards' 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, NextItem will return false (0).

FirstItem ***(Arrayname())***

Executing FirstItem 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())***

Executing LastItem 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()***

Executing PushItem 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()***

PopItem 'pops' or 'recalls' a previously pushed current item pointer for the specified list array. Arrayname() must match the arrayname of the most recently executed PushItem.

ItemStackSize *Max Items*

ItemStackSize determines how many 'list' items may be pushed (using the PushItem command), before items must be 'Pop'ped 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()*

The SortList command is used to rearrange the order of elements in a Blitz linked list. The order in which the items are sorted depends on the first field of the linked list type which must be a single integer word. Sorting criteria will be extended in future releases.

Sort *Arrayname()*

Sort will cause the specified array to be sorted. Only primitive type, 'non-list' arrays may be sorted; it is not possible to sort newtype arrays, or 'list' arrays. 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

SortUp may be used to force the Sort command to sort arrays into ascending order. This means that, after being sorted, an array's contents will be ordered in a 'low to high' manner.

SortDown

SortDown may be used to force the Sort command to sort arrays into descending order. This means that, after being sorted, an array's contents will be ordered in a 'high to low' manner.

R-3: INPUTOUTPUT COMMANDS

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 acheived with the Edit and Print commands, Edit replacing the standard BASIC Input nomenclature. An assortment of commands are available to redirect input and output to and from Files, Windows etc. Refer to the File and Window handling sections for more information.

Those developing games in Blitz should refer to the BlitzIO section for Input Output commands more suited to their particular requirements.

Print *Expression[,Expresion...]*

Print 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[,Expresion...]*

NPrint 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*

Format 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

FloatMode 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 - in exponential format, or in standard format. Exponential format displays a floating point number as a value multiplied by ten raised to a power. For example, 10240 expressed exponentially is displayed as '1.024E+4', ie: 1.024 times 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 floating point 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 floating point 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 your programs. These '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...]*

Read is used to transfer items in Data statements into variables. Data is transferred sequentially into variables through what is known as a 'data pointer'. Each time a pice of data is read, the data pointer is incremented to point at the next piece of data. The data pointer may be set to point to a particular piece of data using the Restore command.

Restore *[Program Label]*

Restore 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 Program Label will become the data to be read when the next Read command is executed. Restore with no parameters will reset the data pointer to the very first piece of data in the program.

Edit$ *([DefaultString$],Characters)*

Edit$ 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 the display. It then waits for the user to hit RETURN. Edit$ returns the text entered by the program user as a string of character.
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)***

Edit is Blitz's standard numeric input command. The same characteristics apply as those for Edit$ however Edit of course only accepts numeric input. program user.

Inkey$ ***[(Characters)]***

Inkey$ may be used to collect one or more characters from the current input channel. The current input channel may be selected using commands such as WindowInput, FileInput or BitMapInput. Inkey$ MAY NOT be used from the DefaultInput input channel as the 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

DefaultInput causes all future Edit$ and Inkey$ functions to receive their input from the CLI window the Blitz program was run from. This is the default input channel used when a Blitz program is first run.

DefaultOutput

DefaultOutput cause all future Print statements to send their output to the 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$,Filename$)*

The FileRequest$ function will open up a standard Amiga-style file requester on the currently used screen. Program flow will halt until the user either selects a file, or hits the requester's 'Cancel' button. If a file was selected, FileRequest$ will return the full 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. Pathname$ MUST be a string variable with a MaxLen of at least 160. Filename$ MUST be a string variable with a MaxLen of at least 64.

PopInput

After input has been re-directed (eg using WindowInput/FileInput), PopInput may be used to return the channel to it's previous condition.

PopOutput

After output has been re-directed (eg using WindowOutput/FileOutput), PopOutput may be used to return the channel to it's previous condition.

Joyx ***(Port)***

Joyx will return the left/right status of a joystick plugged into the specified port. Port must be either 0 or 1, 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)*

Joyy will return the up/down status of a joystick plugged into the specified port. Port must be either 0 or 1, 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 downwrads, Joyy will return 1. If the joystick is held neither upwards or downwards, Joyy will return 0.

Joyr *(Port)*

Joyr may be used to determine the rotational direction of a joystick plugged into the specified port. Port must be either 0 or 1, port 0 beng 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)*

Joyb allows you to read the button status of the device plugged into the specified port. Port must be either 0 or 1, 0 being the port the mouse is normally plugged into. If the left button is held down, Joyb will return 1. If the right button is held down, Joyb will return 2. If both buttons are held down, Joyb will return 3. If no buttons are held down, 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 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 COMMANDS

Blitz supports 2 modes of file access - sequential, and random access. The following section covers the Blitz commands that open, close and operate on 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$)***

OpenFile 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 by OpenFile.
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$)***

ReadFile opens an already existing file specified by Filename$ for sequential reading. If the specified file was successfully opened, ReadFile will return true (-1), otherwise ReadFile will return false (0).
Once a file is open using ReadFile, FileInput may be used to read information from it.

WriteFile ***(File#,Filename$)***

WriteFile 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, WriteFile will return false (0).
A file opened using WriteFile may be written to by using the FileOutput command.

CloseFile ***File#***

CloseFile is 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...]***

Fields is used to set up fields of a random access file record. Once Fields is executed, Get and Put may be 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 initialized 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***

Put is 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***

Get is used to transfer information from a particular record of a random access file into a variable list set up by the Fields command. Only records which also exist may be 'got'.

FileOutput ***File#***

The FileOutput command 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#***

The FileInput command 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 appropraite channel (eg: DefaultInput or WindowInput).

FileSeek ***File#,Position***

FileSeek 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#)***

Lof will return the length, in bytes, of the specified file.

Eof ***(File#)***

The Eof function 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 it's length, while any further reading from the file will cause an error.

Loc ***(File#)***

Loc 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 2048 byte buffer. However, if memory is tight this buffer size may be lowered using the DosBuffLen command.

KillFile ***Filename$***

The KillFile command 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***

ReadMem allows you to read a number of bytes, determined by Length, into an absoulte 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***

WriteMem 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$)***

Exists actually returns the length of the file, unlike Lof() Exists() is for files that have not already been opened. If 0 the file either does not exist or 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$*

ILBMInfo is used to examine an ILBM file. Once ILBMInfo has been executed, ILBMWidth, ILBMHeight and ILBMDepth may be used to examine properties of the image contained in the file.

ILBMWidth

ILBMWidth will return the width, in pixels, of an ILBM image examined with ILBMInfo.

ILBMHeight

ILBMHeight will return the height, in pixels, of an ILBM image examined with ILBMInfo.

ILBMDepth

ILBMDepth will return the depth, in bitplanes, of an ILBM image examined with ILBMInfo.

ILBMViewMode

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. The 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

This function 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

DispHeight will return 256 if executed on a PAL Amiga, or 200 if executed on an NTSC Amiga. This allows programs to open full sized screens, windows etc on any Amiga.

VPos

VPos returns the video's beam vertical position. Useful in both highspeed animation where 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 nember generator in non frame-synced applications.

Peek *[.Type](Address)*

The Peek function 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)*

This function returns the positive equivalent of Expression.

Frac *(Expression)*

Frac() returns the fractional part of Expression.

Int *(Expression)*

This returns the Integer part (before the decimal point) of Expression.

QAbs *(Quick)*

QAbs 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)*

QFrac() returns the fractional part of a quick value. It works like Frac() but accepts a quick value as it's argument. It is faster than Frac() but has the normal quick value limits.

QLimit *(Quick,Low,High)*

QLimit is used to limit the range of a quick number. If Quick is greater than or equal to Low, and less 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)*

QWrap 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)]*

This function returns a random number. If Range is not 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)*

Sgn returns the sign of Expression. If Expression is less than 0, then -1 is returned. If Expression is equal to 0, then 0 is returned. If Expression is greater than 0, then 1 is returned.

Cos *(Float)*

Cos() returns the Cosine of the value Float.

Sin *(Float)*

This returns the Sine of the value Float.

Tan *Tan (Float)*

This returns the Tangent of the value Float.

ACos ***(Float)***

This returns the ArcCosine of the value Float.

ASin ***(Float)***

This returns the ArcSine of the value Float.

ATan ***(Float)***

This returns the ArcTangent of the value Float.

HCos ***(Float)***

This returns the hyperbolic Cosine of the value Float.

HSin ***(Float)***

This returns the hyperbolic Sine of the value Float.

HTan ***(Float)***

This returns the hyperbolic Tangent of the value Float.

Exp ***(Float)***

This returns e raised to the power of Float.

Sqr ***(Float)***

This returns the square root of Float.

Log10 ***(Float)***

This returns the base 10 logarithm of Float.

Log ***(Float)***

This returns the natural (base e) logarithm of Float.

QAngle ***(Src X,Src Y,Dest X,Dest Y)***

QAngle returns the angle between the two 2D coordinates passed. the angle.q returned is a value from 0..1, 1 representing 360 degrees in standard polar geometry.

Left$** ***(String$,Length)*

This function returns Length leftmost characters of string String$.

Right$** ***(String$,Length)*

Right$() returns the rightmost Length characters from string String$.

Mid$** ***(String$,Startchar[,Length])*

This function 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)***

Hex$() returns an 8 character string equivalent to the hexadecimal representation of Expression.

Bin$ ***(Expression)***

Hex$() returns a 32 character string equivalent to the binary representation of Expression.

Chr$ ***(Expression)***

Chr$ 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$)***

Asc() returns the ASCII value of the first characters in the string String$.

String$** ***(String$,Repeats)*

This function will return a string containing Repeats sequential occurrences of the string String$.

Instr ***(String$,Findstring$[,Startpos])***

Instr attempts to locate FindString$ within String$. If a match is found, the character position of the first matching character will be returned. If no match is found, 0 will be returned.
The optional Startpos parameter allows you to specify a starting character position for the search.
CaseSense may be used to determine whether the search is case sensitive or not.

Replace$** ***(String$,Findstring$,Replacestring$)*

Replace$() will search the string String$ for any occurrences of the string Findstring$ and replace it with the string Replacestring$.
CaseSense may be used to determine whether the search is case sensitive or not.

Mki$ ***(Integer)***

This will create a two byte character string, given the two byte numeric value Numeric. Mki$ is often used before writing integer values to sequential files to save on disk space. When the file is later read in, Cvi may be used to convert the string back to an integer.

Mkl$ ***(Long)***

This will create a four byte character string, given the four byte numeric value Long. Mkl$ is often used when writing long values to sequential files to save on disk space. When the file is later read in, Cvl may be used to convert the string back to a long.

Mkq$ ***(Quick)***

This will create a four byte character string, given the four byte numeric value Quick. Mkq$ is often used when writing quick values to sequential files to save on disk space. When the file is later read in, Cvq may be used to convert the string back to a quick.

Cvi ***(String$)***

Cvi returns an integer value equivalent to the left 2 characters of String$. This is the logical opposite of Mki$.

Cvl ***(String$)***

Cvl returns a long value equivalent to the left 4 characters of String$. This is the logical opposite of Mkl$.

Cvq ***(String$)***

Cvq returns a quick value equivalent to the left 4 characters of String$. This is the logical opposite of Mkq$.

Len ***(String$)***

Len returns the length of the string String$.

UnLeft$** ***(String$,Length)*

UnLeft$() removes the rightmost Length characters from the string String$.

UnRight$** ***(String$,Length)*

UnRight$() removes the leftmost Length characters from the string String$.

StripLead$** ***(String$,Expression)*

StripLead$ removes all leading occurrences of the ASCII character specified by Expression from the string String$.

StripTrail$** ***(String$,Expression)*

StripTrail$ removes all trailing occurrences of the ASCII character specified by Expression from the string String$.

LSet$** ***(String$,Characters)*

This function returns a string of Characters characters long. The string String$

will be placed at 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)*

This function 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)*

This function returns a string of Characters characters long. The string String$ will be centered 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.

LCase$** ***(String$)*

This function returns the string String$ converted into lowercase.

UCase$** ***(String$)*

This function returns the string String$ converted to uppercase.

CaseSense ***On| Off***

CaseSense allows you to control 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$)***

This functions 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)***

This 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)***

This 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

SystemDate returns the system date as the number of days passed since 1/1/1978.

Date$ *(days)*

Date$ converts the format returned by SystemDate (days passed since 1/1/1978) into a string format of dd/mm/yyyy or mm/dd/yyyy depending on the dateformat (defaults to 0).

NumDays *(date$)*

Numdays 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*

DateFormat 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

The function Processor returns the type of processor in the computer on which the program is currently running.

0=68000
1=68010
2=68020
3=68030
4=68040

ExecVersion

The function 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 speicalised data concerned with windows, shapes etc.

USEPATH ***Pathtext***

USEPATH 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

The BLITZ directive is used to enter Blitz mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz Programmers Guide.

AMIGA

The AMIGA directive is used to enter Amiga mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz Programmers Guide.

QAMIGA

The QAMIGA directive is used to enter Quick Amiga mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz Programmers Guide.

INCLUDE ***Filename***

INCLUDE is 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 (ie: 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*

XINCLUDE stands for exclusive include. XINCLUDE works identically to INCLUDE with the exception that XINCLUDE'd files are only ever included once. For example, if a program has 2 XINCLUDE statements with the same filename, only the first XINCLUDE will have any effect.
INCDIR may be used to specify a path for Filename.
Filename may be optionally quote enclosed to avoid tokenisation problems.

INCBIN *Filename*

INCBIN 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.
INCDIR may be used to specify an AmigaDos path for Filename.
Filename may be optionally quote enclosed to avoid tokenisation problems.

INCDIR *Pathname*

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.
Pathname may be optionally quote enclosed to avoid tokenisation problems.

CNIF *Constant Comparison Constant*

CNIF allows you to conditionally compile a section of program code based on a comparison of 2 constants. Comparison should be one of '<', '>', '=', '<>', '<=' or '>='. 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.

CEND

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"*

CSIF allows you to conditionally compile a section of program code based on a comparison of 2 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

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***

CERR 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***

Macro is 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

End Macro is 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#***

Use will cause the Blitz object specified by Objectname and Object# to become the currently used object.

Free ***Objectname Object#***

Free is used to free a Blitz object. Any memory consumed by the object's existance will be free'd up, and in the case of things such as windows and screens, the display may be altered.
Attempting to free a non-existant object will have no effect.

USED ***ObjectName***

Used returns the currently used object number. This is useful for routines which need to operate on the currently used object, also interrupts should restore currently used object settings.

Addr ***Objectname(Object#)***

Addr is a low-level function allowing advanced programmers the ability to find where a particular Blitz object resides in RAM. An appendix at the end of this manual lists all Blitz object formats.

Maximum ***Objectname***

The Maximum function 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 it's built in assembler. This allows the programmer to include machine code in their programs. Those familiar with 68000 assembler will find the ability to mix easy to code BASIC with their own lightning fast machine code routines a powerful concoction.

There are three ways of including assembler in Blitz programs:

Inline: using PutReg 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 the case of Functions the value in D0 is returned to the caller. The AsmExit command is used in place of StatementReturn or FunctionReturn.

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 assmbler 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***

DCB stand for 'define constant block'. DCB allows you to insert a repeating series of the same value into your assembler programs.

EVEN

EVEN allows you 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 ***68000 Reg,Expression***

GetReg 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.
GetReg should only be used to transfer expressions to one of the 8 data registers (d0-d7).
GetReg will use the stack to temporarily store any registers used in calculation of the expression.

PutReg ***68000 Reg,Variable***

PutReg may be 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***

SysJsr allows you to call any of Blitz's system routines from your own program. Routine specifies a routine number to call.

TokeJsr ***Token[,Form]***

TokeJsr allows you 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. A full list of all token numbers with their various forms will be available shortly from Acid Software.

ALibJsr ***Token[,Form]***

ALibJsr 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 programmers guide for more information on library writing.

BLibJsr ***Token[,Form]***

BLibJsr 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 programmers guide for more information on library writing.

AsmExit

AsmExit is used to exit from functions and statements written in assembler. Remember also that registers A4-A6 must be preserved in functions and statements written in assembler.

R-8: MEMORY CONTROL COMMANDS

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*

The Poke command will place the specified Data into the 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, you would use Poke.b; to poke a word into memory you would use Poke.w; and to poke a long word into memory you would 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)*

The Peek function 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)*

Peeks$ will return a string of characters corresponding to bytes peeked from consective memory locations starting at Address, and Length characters in length.

Call *Address*

Call will cause program flow to be transferred to the absolute memory location specified by Address.
PLEASE NOTE! 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.
Please refer to the 'Assembler' section of the manual for the rules machine code programs must follow to operate correctly within the Blitz environment.

Bank ***(Bank#)***

Returns the memory location of the given memory Bank, replaces the older and more stupidly named BankLoc command.

BankSize ***(Bank#)***

BankSize returns the size of the memory block allocated for the specified Bank#.

InitBank ***Bank#,size,memtype***

InitBank 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#***

FreeBank de-allocates any memory block allocated for the Bank specified.

LoadBank Bank#,FileName$[,MemType]

The LoadBank command has been modified, instead of having to initialise the bank before loading a file, LoadBank 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$***

SaveBank will 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 COMMANDS

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

The NumPars function 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:

myprog file1 file2

In this case, NumPars would return the value '2' - 'file1' and 'file2' beng 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 it's '.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 Arguement' menu item in the 'COMPILER' menu allows you to test out CLI parameters.

Par$ *(Parameter)*

Par$ return a string equivalent to a parameter passed to an executable file through either the CLI or Workbench. Please refer to NumPars for more information on parameter passing.

CloseEd

The CloseEd statement will cause 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 about 40K of chip memory.
CloseEd will have no effect on executable files run outside of the Blitz environment.

NoCli

NoCli will prevent the normal 'Default Cli' from opening when programs are executed from within Blitz. NoCli has no effect on executable files run outside of the Blitz environment.

FromCLI

Returns TRUE (-1) if your program was run from the CLI, or FALSE (0) if run from the WorkBench.

ParPath$ *(parameter,type)*

ParPath$ returns the path that the parameter resides in, '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/myprogs" then ParPath$(0,0) willreturn "work:mystuff/myprograms", but ParPath$(0,1) will return "work:mystuff/myprograms/FRED".
The way WB 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: SLICE COMMANDS

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 lice). They may not be positioned side by side.

Slice *Slice#,Y,Flags*
Slice#,Y,Width,Height,Flags,BitPlanes,Sprites,Colours,w1,w2

The Slice command is 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 eight 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.
Width1 and Width2 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, Width1 refers to the width of the 'foreground' bitmap, and Width2 refers to the width of the 'background' bitmap. If a slice is NOT set up to be a dual-playfield slice, both Width1 and Width2 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
$fff8A	Standard lo-res slice	6
$fff9A	Standard hi-res slice	4
$fffaA	Lo-res, dual-playfield slice	6
$fffbA	Hi-res, dual-playfiled slice	4
$fffcA	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 numbers 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 allowins 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#*

Use Slice is used to set the specified slice object as being the currently used slice. This is required for commands such as Show, ShowF, ShowB and Blitz mode RGB.

FreeSlices

FreeSlices is used to completely 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]*

Show is 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 it's 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]]*

ShowF is 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 it's top-left. Omitting the X and Y parameters is identical to supplying X and Y 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]]*

ShowB is 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 it's top-left. Omitting the X and Y parameters is identical to supplying X and Y 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 *Colour Register,Red,Green,Blue,Y*

ColSplit 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*

CustomCop 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

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

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

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*

Display 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*

The SetBPLCON0 command has been added for advanced control of Slice display modes. The BPLCON0 hardware register is on page A4-1 of the reference manual (appendix 4). 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

R-11: DISPLAY LIBRARY COMMANDS

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 commanddoes 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 disapointed to hear that their displays must be modified to some 256 pixels across.

The way the Amiga fetches data for each scanline 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*

InitCopList is 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!) colors (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	$10
#dualplayfield	$20
#extrahalfbrite	$40
#ham	$80
#lores	$000
#hires	$100
#super	$200
#loressprites	$400
#hiressprites	$800
#supersprites	$c00
#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#..]***

CreateDisplay is 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]]***

The DisplayBitMap 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 for dualplayfield.

DisplaySprite ***CopList#,Sprite#,X,Y,Sprite Channel***

DisplaySprite 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]***

DisplayPalette copies colour information from a Palette to the CopList specified.

DisplayControls *CopList#,BPLCON2,BPLCON3,BPLCON4*

DisplayControls 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&^)

The default values are given at the top of the table, the 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** ($224)	**BPLCON3** ($c00)	**BPLCON4** ($11)
15	*	BANK2 * activecolorbank	BPLAM7 ;xorithbitplane
14	ZDBPSEL2	BANK1 *	BPLAM6 ;DMA altering
13	ZDBPSEL1	BANK0 *	BPLAM5 ;effectivecolour
12	ZDBPSEL0	PF2OF2 coloffset pfield 2	BPLAM4 ;look up
11	ZDBPEN	PF2OF1	BPLAM3
10	ZDCTEN	PF2OF0	BPLAM2
09	KILLEHB *	LOCT *palette hi/lo nibble	BPLAM1
08	RDRAM=0 *		BPLAM0
07	SOGEN	SPRES1 *sprite res	ESPRM7 high order color
06	PF2PRI H	SPRES0 *	ESPRM6 offset for even
05	PF2P2	BRDRBLANK border	ESPRM5 sprites
04	PF2P1	BRDNTRAN zd=border	ESPRM4
03	PF1P0		OSPRM7 hiorder color
02	PF1P2	ZDCLCKEN zd=14mhz	OSPRM6 offset for odd
01	PF1P1	BRDSPRT spritesinborders!	OSPRM5 sprites
00	PF1P0	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 widthadjust 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 Commodores AGA circuitry. Ha ha ha. No to be quite serious I really do not have a clue how they cludeged 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 Jaguarlib for Blitz. (close than you think)...

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 customcops 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 that large AGA palette changes may take several lines of the display to be complete.

CustomString ***CopList#,CCOffset,YPos,Copper$***

CustomString 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 the 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***

CustomSprites 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***

DisplayDblScan is 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]***

DisplayRainbow is used to alter a certain colour register vertically down a display. It simple maps each colour in a palette to the coresponding vertical position of the display. ECS displays require one copper instruction per line while AGA displays require 4.

DisplayRGB ***CopList#,Register,line,r,g,b[,copoffset]***

DisplayRGB 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]*

DisplayUser 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]*

DisplayScroll 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 hiresolution positioning of the bitmap. Three instructions per line are required for the DisplayScroll command.

R-12: BLITZMODE IO COMMANDS

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*

BlitzKeys is 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

BlitzQualifier returns any qualifier keys that were held down in combination with the last inkey$ during BlitzMode input.

BlitzRepeat *Delay,Speed*

BlitzRepeat 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 will the Blitz mode keyboard reading is enabled. This is done using the BlitzKeys command.

RawStatus *(Rawkey)*

The RawStatus 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*

The Mouse command turns on or off Blitz mode's ability to read the mouse. Once a Mouse On has been executed, programs can read the mouse's position or speed in Blitz mode.

Pointer *Sprite#,Sprite Channel*

The Pointer command 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 the 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*

MouseArea 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 using 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 a Mouse On command, the MouseY function may be using 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 interupt.

MouseYSpeed

If Blitz mode mouse reading has been enabled using a Mouse On command, the MouseYSpeed function may be using 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 interupt.

LoadBlitzFont *BlitzFont#,Fontname.font$*

LoadBlitzFont 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 LoadBlitzFont 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 eight by eight non-proportional fonts.

Use BlitzFont *BlitzFont#*

If you have loaded two or more blitzfont objects using LoadBlitzFont, Use BlitzFont may be used to select one of these fonts for future bitmap output.

Free BlitzFont *BlitzFont#*

Free BlitzFont 'unloads' a previously loaded blitzfont object. This frees up any memory occupied by the font.

BitMapOutput *BitMap#*

BitMapOutput 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 eight by eight non-proportional fonts.
BitMapOutput is mainly of use in Blitz mode, as other forms of character output become unavailable in Blitz mode.

Colour *Foreground Colour[,Background Colour]*

Colour allows you to alter the colours use to render text to bitmaps. Foreground colour allows you to specify the colour text is rendered in, and the optional Background colour 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 eigth.
Y specifies a character position down the bitmap, and may be a fractional value. For example, a Y of 1.5 will set a cusor 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

BitMapInput 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. A BitMapOutput must also be executed before any Edit$ or Edit commands are encountered.

R-13: BITMAP COMMANDS

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*

BitMap creates and initializes 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#*

Use 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#*

Free 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#*

CopyBitMap will make 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#*

Blitz 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#]***

LoadBitMap allows you to load an ILBM IFF graphic into a previously initialized 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#]***

SaveBitMap 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***

BitPlanesBitMap 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#***

ShapesBitMap creates a dummy BitMap so drawing commands can be used directly on a shapes image data.

CludgeBitMap ***BitMap#,Width,Height,Depth,Memory***

CludgeBitMap will create 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 commands 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***

BitMapWindow 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 ***BitMapOrigin BitMap#,x,y***

BitMapOrigin allows the programmer to relocate the origin (0,0) of the bitmap used by the drawing commands line, poly, box and circle.

DecodeILBM ***DecodeILBM BitMap#,MemoryLocation***

A very fast method of unpacking standard iffilbm 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.

R-14: 2D DRAWING COMMANDS

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]*

Cls 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*

Plot is used to alter the colour of an individual pixel on the currently used bitmap. X and Y specify the location of the pixel to be altered, and Colour specifies the colour to change the pixel to. A Colour parameter of -1 will cause the pixel at the specified pixel position to be 'inverted'.

Point *(X,Y)*

The Point function will 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*

The Line command 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, 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*

The Box command draw a rectangular outline onto the currently used bitmap. 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*

Boxf draws a solid rectangular shape on the currently used bitmap. 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[,Y Radius],Colour*

Circle will draw an open circle onto the currently used bitmap. X and Y specify the mid point of the circle. The Radius parameter specifies the radius of the circle. If a Y Radius 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[,Y Radius],Colour***

Circlef will draw a filled circle onto the currently used bitmap. X and Y specify the mid point of the circle - Colour, the colour in which to draw the circle. The Radius parameter specifies the radius of the circle. If a Y Radius 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[,Source BitMap]***

Scroll allows rectangular areas within a bitmap to be moved around. X1, Y1, Width and Height specify the position and size of the rectangle to be moved. X2 and Y2 specify the position the rectangle is to be moved to.
An optional Source BitMap parameter allows you to move rectangular areas from one bitmap to another.

FloodFill ***X,Y,Colour [,Border Colour]***

FloodFill 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

FreeFill will deallocate 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 to do it's 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]***

ReMap is 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,color***

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 verticies are listed in anti-clockwise order.

R-15: ANIMATION SUPPORT COMMANDS

The following 4 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#]*

The LoadAnim command will create an Anim object and load a DPaint compatible animation. The ILBMInfo command can be used to find the correct screensize 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#]*

InitAnim 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#*

NextFrame renders the nextframe 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#)*

The Frames() function returns the number of frames in the specified Anim.

R-16: SHAPE HANDLING COMMANDS

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 animbrushes use the LoadShape/SaveShape commands and the faster Blitz format uses the LoadShapes and SaveShapes format.

LoadShape *Shape#,Filename$[,Palette#]*

LoadShape 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.
The LoadShape 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#*

SaveShape will create 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$*

LoadShapes 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 the 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$*

SaveShapes 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*

GetaShape lets you transfer a rectangular area of the currently used bitmap into the specified shape object. X, Y, Width and Height specify the area of the bitmap to be picked up and used as a shape.

CopyShape *Shape#,Shape#*

CopyShape will produce 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 do, however, consume some of your valuable Chip memory.
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.
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#*

MakeCookie 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#)*

The ShapeWidth function returns the width, in pixels, of a previously created shape object.

ShapeHeight *(Shape#)*

The ShapeHeight function 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 'acrosswards' offset for a handle, the Y

parameter specifies a 'downwards' offset.
Let's have a look at an example of how a handle works. Assume you have set a shapes X handle to 5, and it's 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 actually 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, it's handle is automatically set to 0,0 - it's upper left corner.

MidHandle ***Shape#***

MidHandle will cause the handle of the specified shape to be set to it's 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#***

The XFlip command is 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#***

The YFlip command may be used to vertically 'mirror' a shape object. The resultant shape will appear to have been 'turned upside down'.

Scale ***Shape#,X Ratio,Y Ratio[,Palette#]***

Scale is a very powerful command which may be used to 'stretch' or 'shrink' shape objects. The Ratio 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#,Angle Ratio***

The Rotate command allows you to rotate a shape object. Angle Ratio specifies how much clockwise rotation to apply, and should be in the range zero to one. For instance, an Angle Ratio of .5 will cause a shape to be rotated 180 degrees, while an Angle Ratio of .25 will cause a shape to be rotated 90 degrees clockwise.

DecodeShapes ***Shape#[,Shape#],MemoryLocation***

DecodeShapes, 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***

InitShape 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 COMMANDS

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 animations 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 it's 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]*

Blit is 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*

The BlitMode command 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 and their effects:

CookieMode:	Shapes are drawn 'as is'.
EraseMode:	An area the size and shape of the shape willbe 'erased' on the destination bitmap.
InvMode:	An area the size and shape of the shape willbe 'inversed' 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

The CookieMode function returns a value which may be used by one of the commands involved in blitting modes.
Using CookieMode as a blitting mode will cause a shape to be blitted cleanly, 'as is', onto a bitmap.

EraseMode

The EraseMode 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

The InvMode 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#,Max Items*

The Queue command 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 it's new location on the display
Step 5: If there are any more objects to move, go 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. Lets have a 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 it's 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 initialize 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 Max Items parameter.

QBlit ***Queue#,Shape#,X,Y[,Excessonoff]***

QBlit 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.
Please refer to the Queue command for an explanation of the use of queues.
The optional Excessonoff parameter works identically to the Excessonoff parameter used by the Blit command. Please refer to the Blit command for more information on this parameter.

UnQueue ***Queue#[,BitMap#]***

UnQueue is used to erase all 'remembered' items in a queue. Items are placed in a queue by use of the QBlit command. Please refer to Queue for a full explanation of queues and their usage.
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#***

FlushQueue will force the specified queue object to be 'emptied', causing the next UnQueue command to have no effect.

QBlitMode ***BLTCON0***

QBlitMode allows you to control how the blitter operates when QBlitting shapes to bitmaps. Please refer to BlitMode for more information on this command.

Buffer *Buffer#,Memorylen*

The Buffer command is 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 the Buffer 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 BBlited, and the maximum number of shapes to be BBlited 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 initialized buffer. For more information on how buffers work, please refer to the Buffer command.
The optional Excessonoff parameter works identically to the Excessonoff parameter used by the Blit command. Please refer to the Blit command for more information on this parameter.

UnBuffer *Buffer#*

UnBuffer is 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#*

FlushBuffer will force the specified buffer object to be 'emptied', causing the next UnBuffer command to have no effect.

BBlitMode *BLTCON0*

BBlitMode allows you to control how the blitter operates when BBlitting shapes to bitmaps. Please refer to BlitMode for more information on this command.

Stencil ***Stencil#,BitMap#***

The Stencil command will create 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]***

SBlit 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***

SBlitmode is used to determine how the SBlit command operates. Please refer to the BlitMode command for more information on blitting modes.

ShowStencil ***Buffer#,Stencil#***

ShowStencil is 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 BBlitted, executing the ShowStencil command will repair the damage done to the foreground graphics!

Block ***Shape#,X,Y***

Block is 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 that the height and y destination of the shape are not limited by the Block command.
Block is intended for use with map type displays.

BlitColl ***(Shape#,x,y)***

BlitColl is 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 course1bcan not tell you what shapeyou have collided with.

ClipBlit *ClipBlit Shape#,X,Y*

ClipBlit is the 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, X2 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 COMMANDS

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 it's 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 anymore. 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,Sprite Channel*

ShowSprite is the command used to actually display a sprite through a sprite channel. X and Y specify the position the sprite is to be displayed at. These parameters are ALWAYS given in lo-resolution pixels. Sprite Channel is a value 0 through 7 which decides which sprite channel the sprite should be display through.

InFront *Sprite Channel*

A feature of sprites is that they may be displayed either 'in front of' or

'behind' the bitmap graphics they are appearing in. The InFront command allows you to determine which sprites appear in front of bitmaps, and which sprites appear behind.
Sprite Channel 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 Sprite Channel will appear BEHIND any bitmap graphics. Sprites displayed through channels less than Sprite Channel 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 ***Sprite Channel***

InFrontF is 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, some between the playfields, and some behind both playfields.

InFrontB ***Sprite Channel***

InFrontB is used on dualplayfield slices to determine sprite/playfield priority with respect to the background playfield. Using combinations of InFrontB and InFrontF (used for 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$***

LoadSprites 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$***

SaveSprites 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 correspong to the widths 16, 32 and 64.

R-19: COLLISION DETECTION COMMANDS

This section deals with various commands involved in the 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 'collidable' colour. Bitplanes refers to the number of bitplanes (depth) of the bitmap collisions are to be tested for 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

SetCollOdd is 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 (ie: 1,3,5...). This allows you to design bitmap graphics in such a way that even coloured areas are 'safe' (ie: they will not report a collision) whereas odd colour areas are 'unsafe' (ie: they will report a collision).
The DoColl and PColl commands are used to detect the actual sprite/bitmap collisions.

SetCollHi *BitPlanes*

SetCollHi 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

DoColl is 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 ***(Sprite Channel)***

The PColl function may be used to find out if a particular sprite has collided with any bitmaps. Sprite Channel refers to the sprite channel 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. Please refer to DoColl for more information.

SColl ***(Sprite Channel, Sprite Channel)***

SColl may be 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)***

The ShapesHit 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)***

The ShapeSpriteHit 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)***

The SpritesHit 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)***

The RectsHit 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: PALETTE COMMANDS

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$[,Palette Offset]*

LoadPalette creates and initializes 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 Palette Offset 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#*

ShowPalette replaces Use Palette for copying a palette's colours to the current Screen or Slice.

Use Palette *Palette#*

Use 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*

The NewPaletteMode 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#*

Free 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 colors.

CyclePalette ***Palette#***

CyclePalette uses the standard color cycling parameters in the palette object to cycle the colors. 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***

FadePalette multiplies all colours in a Palette by the Brightness argument and places the result in the DestPalette.

InitPalette ***Palette#,NumColors***

InitPalette simply initialises a palette object to hold NumColors. All colors will be set to black.

DecodePalette ***Palette#,MemoryLocation[,Palette Offset]***

DecodePalette allows the programmer to unpack included iff palette information to Blitz palette objects.

PalRGB ***Palette#,Colour Register,Red,Green,Blue***

PalRGB 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 Use Palette is executed.

RGB ***Colour Register,Red,Green,Blue***

RGB 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 ***(Colour Register)***

Red 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.
Red will always return a value in the range zero to fifteen.

Green ***(Colour Register)***

Green 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. Green will always return a value in the range zero to fifteen.

Blue ***(Colour Register)***

Blue 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. Blue will always return a value in the range zero to fifteen.

AGARGB ***Colour Register,Red,Green,Blue***

The AGARGB command is the AGA equivalent of the RGB command. The 'Red', 'Green' and 'Blue' parameters must be in the range 0 through 255, while 'Colour Register' is limited to the number of colours available on the currently used screen.

AGAPalRGB ***Palette#,Colour Register,Red,Green,Blue***

The AGAPalRGB command 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 'Use Palette' is executed.

AGARed ***(colour register)***

The AGARed function 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 ***(colour register)***

The AGAGreen function 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 ***(colour register)***

The AGABlue function 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,Low Colour,High Colour [,Speed]***

SetCycle is used to configure colour cycling information for the Cycle command. The 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#*

Cycle will cause 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

StopCycle will halt all colour cycling started with the Cycle command.

FadeIn *Palette#[,Rate[,Low Colour, High Colour]]*

Fadein will cause the colour palette of the currently used slice to be 'faded in' from black up to the RGB values contained in the specified Palette#.
Rate# allows you to control the speed of the fade, with 0 being the fastest fade.
Low Colour and High Colour allow you to control which colour palette registers are affected by the fade.

FadeOut *Palette#[,Rate[,Low Colour, High Colour]]*

Fadeout will cause the colour palette of the currently used slice to be 'faded out' from the RGB values contained in the specified Palette# down to black.
Rate# allows you to control the speed of the fade, with 0 being the fastest fade.
Low Colour and High Colour allow you 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*

ASyncFade 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 thewhole 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

DoFade will cause 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 any steps of fading left to perform.

FadeStatus

FadeStatus is 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***

PaletteRange creates a spread of colors within a palette. Similar to DPaint's spread function PaletteRange takes a start and end colour and creates the color tweens between them.

DuplicatePalette ***SrcPalette#,DestPalette#***

DuplicatePalette simply creates a new Palette which exactly matches the SrcPalette.

R-21: SOUND MUSIC & SPEECH COMMANDS

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 medmodule 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$*

LoadSound 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...]]*

Sound 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 one through fifteen.

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...]]***

LoopSound 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...]***

Volume 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]]***

InitSound initializes 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***

SoundData 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)***

PeekSound returns the byte of a sample at the specified offset of the sound object specified.

DecodeSound ***Sound#,MemoryLocation***

DecodeSound, similar to the other new Decode commands allows the programmer to include sound files within their program's object code.

SetPeriod ***Sound#,Period***

This command allows the programmer to manually adjust the period of the sound object to change it's effective pitch.

DiskPlay ***Filename$,Channelmask[,Vol1[,Vol2...]]***

DiskPlay will play 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. The DiskPlay 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. Also, DiskPlay allows you to play samples of any length, whereas LoadSound only allows samples up to 128K in length to be loaded.

DiskBuffer ***Bufferlen***

DiskBuffer 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$***

LoadModule loads in from disk a soundtracker/noisetracker music module. This module may be later played back using PlayModule.

Free Module ***Module#***

Free Module may be used to delete a module object. Any memory occupied by the module will also be free'd.

PlayModule ***Module#***

PlayModule will cause a previously loaded soundtracker/noisetracker song module to be played back.

StopModule

StopModule will cause any soundtracker/noisetracker modules which may be currently playing to stop.

LoadMedModule *MedModule# Name*

The LoadMedModule command loads any version 4 channel Octamed module. The following routines support upto 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 AmigaMode.

StartMedModule *MedModule#*

StartMedModule is 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

PlayMed is responsible for playing the current MedModule, it must be called every 50th of a second either on an interupt (#5) or after a VWait in a program loop.

StopMed

StopMed will cause any med module to stop playing. This not only means that PlayMed will have no affect until the next StartMedModule but silences the audio channels so they are not left ringing as is the effect when PlayMed is not called every vertical blank.

JumpMed *Pattern#*

JumpMed will change the pattern being played in the current module.

SetMedVolume *Volume*

SetMedVolume 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#*

GetMedVolume returns the current volume setting of the specified audio channel. This is useful for graphic effects that you may wish to sync to certain channels of the music playing.

GetMedNote *Channel#*

GetMedNote 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***

GetMedInstr returns the current instrument playing through the specified audio channel.

SetMedMask ***Channel Mask***

SetMedMask allows the user to mask out audio channels needed by sound effects stopping the Med Library using them.

DecodeMedModule ***MedModule#,MemoryLocation***

DecodeMedModule replaces the cludgemedmodule, as med modules are not packed but used raw, DecodeMedModule simply checks to see the memorylocation passed is in ChipMem (if not it copies the data to chip) and points the Blitz MedModule object to that memory.

Speak ***string$***

The Speak command will first convert the given string to phonetics and then pass it to the 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***

SetVoice alters the sound of the Amiga's speech synthsiser by changing the vocal characteristics listed in the parameters above.

Translate$** ***(string$)*

Translate$() returns the phonetic equivalent of the string for use with the Translate

PhoneticSpeak ***phonetic$***

PhoneticSpeak is similar to the Speak command but should only be passed strings containing legal phonemes such as that produced by the Translate$() function.

VoiceLoc

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 trebly sounds.

R-22: SCREEN COMMANDS

The following section covers the Blitz commands that let you open and control Intuition based Screen objects.

Command **Description**

Screen ***Screen#,Mode[,Title$]***
Screen#,X,Y,Width,Height,Depth,VMode,Title$,Dpen,Bpen[,BMap#]

Screen will open an Intuition screen. The are 2 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 screenmode you require:
hires=$8000
ham=$200
superhires=$20
interlace=4
lores=0

ShowScreen ***Screen#***

ShowScreen will cause the specified screen object to be moved to the front of the display.

WbToScreen ***Screen#***

WbToScreen will assign 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. It's 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#]

LoadScreen 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 Use Palette statement.

SaveScreen ***Screen#,Filename$***

SaveScreen will save 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

SMouseX returns the horizontal position of the mouse relative to the left edge of the currently used screen.

SMouseY

SMouseY returns the vertical position of the mouse relative to the top of the current screen.

ViewPort ***(Screen#)***

The ViewPort function returns the location of the specified screens ViewPort. The ViewPort address can be used with graphics.library commands and the like.

ScreenPens ***active text, inactive text, hilight, shadow, active fill, gadget fill***

ScreenPens configures the 10 default pens used for system gadgets in WorkBench 2.0. 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#***

CloseScreen 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 Screen to back of all Screens open in the system.

BeepScreen ***Screen#***

Flash specified screen.

MoveScreen ***Screen#,deltax,deltay***

Move specified screen by specified amount. Good for system friendly special effects.

ScreenTags ***Screen#,Title$ [&TagList] or [[,Tag,Data]...]***

Full access to all the Amiga's new display resoutions is now available in Amiga mode by use of the Screen Tags command. The following tags are of most interest to Blitz programmers: (see autodocs/

```
#Left=$80000021:#Top=$80000022:#Width=$80000023
#Height=$80000024:#Depth=$80000025:#DetailPen=$80000026
#BlockPen=$80000027
#Title=$80000028:#Colors=$80000029:#ErrorCode=$8000002A
#Font=$8000002B:#SysFont=$8000002C:#Type=$8000002D
#BitMap=$8000002E
#PubName=$8000002F:#PubSig=$80000030
#PubTask=$80000031:#DisplayID=$80000032
#DClip=$80000033:#Overscan=$80000034
#ShowTitle=$80000036:#Behind=$80000037:#_Quiet=$80000038
#AutoScroll=$80000039:#Pens=$8000003A
#FullPalette=$8000003B:#ColorMapEntries=$8000003C
#Parent=$8000003D:#Draggable=$8000003E
#Exclusive=$8000003F
#SharePens=$80000040:#BackFill=$80000041
#_Interleaved=$80000042
#Colors32=$80000043:#VideoControl=$80000044
#FrontChild=$80000045:#BackChild=$80000046
#LikeWorkbench=$80000047:#Reserved=$80000048
```

ShowBitMap ***[BitMap#]***

The ShowBitMap command is the Amiga-mode version of the Show command. It enables you to change a Screens 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: WINDOW COMMANDS

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.

Command **Description**

Window ***Window#,X,Y,Width,Height,Flags,Title$,Dpen,Bpen[,GadgetList#]***

Window opens an Intuition window on the currently used screen. Window# is a unique object number for the new window. X and Y refer to the offset from the 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, dragbar 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 dragbar 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 the 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 it's 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 window and allows it to be sized.
WINDOWDRAG	$0002	Allows window to be dragged with the mouse by it's title bar.
WINDOWDEPTH	$0004	Lets windows be pushed behind or pulled in front of other windows.
WINDOWCLOSE	$0008	Attaches a closegadget to the upper left corner of the window.
SIZEBRIGHT	$0010	With GIMMEZEROZERO and WINDOWSIZING set, this will leave the right hand margin, the width of the sizing gadget, clear, and any drawing to the 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 convevient, 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#*

Use Window will cause 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#*

Free 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#*

WindowInput will cause 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 it's input from somewhere else useful (for example, using another WindowInput command) before executing another Inkey$, Edit$ or Edit function.

WindowOutput *Window#*

WindowOutput will cause 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.

DefaultIDCMP *IDCMP_Flags*

DefaultIDCMP 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 it's 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 occured.
$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 *IDCMP_Flags*

AddIDCMP 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 *IDCMP_Flags*

SubIDCMP 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

WaitEvent will halt program excution 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 occured.
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

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 occured, Event will return 0.

EventWindow

EventWindow may be used to determine in which window the most recent window event occured. Window events are detected by use of either the WaitEvent or Event commands.
EventWindow return the window object number in which the most recent window event occured.

FlushEvents *[IDCMP_Flag]*

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 occured between executions of WaitEvent or Event will be stored in a queue for later use. However, there may be situations where you want to ignore this backlog of events. This is what FlushEvents is for.
Executing FlushEvents with no parameters will completely clear Blitz's internal event queue, leaving you with no outstanding events. Supplyng an IDCMP_Flag parameter will only clear events of the specified type from the event queue.

GadgetHit

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 posess the same identification numbers, you may also need to use EventWindow to tell exactly which gadget was hit.

MenuHit

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 the 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

ItemHit returns the identification number of the menu item that caused the last menu event.

SubHit

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

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

RawKey returns the raw key code of a key that caused the most recent 'key press' event.

Qualifier

Qualifier will return 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*

WPlot 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*

WBox 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*

WCircle allows you to draw a circle in the 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,X Radius,Y Radius,Colour***

WEllipse draws an ellipse in the currently used window. You specify the centre of the ellipse with the coordinates X,Y. X Radius specifies the horizontal radius of the ellipse, Y Radius 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 to be drawn.

WCls ***[Colour]***

WCls will clear the currently used window to clour 0, or 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]***

InnerCls will clear only the inner portion of the currently used window. It will not clear the titlebar or borders as Cls would do if your window was not opened with the GIMMEZEROZERO flag set. If colour is specfied, then that colour will be used to clear the window.

WScroll ***X1,Y1,X2,Y2,Delta X,Delta Y***

WScroll will 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***

Cursor will 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 *[Characterpos]*

EditFrom allows you to control how the Edit$ and Edit functions operate when used within windows.
If a Characterpos parameter is specified, then the next time an edit function is executed, editting will commence at the specified character position (0 being the first character position).
Also, editting may be terminated not just by the use of the 'return' key, but also by any non printable character (for example, '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 Characterpos is omitted, Edit$ and Edit return to normal - editting always beginning at character postition 0, and 'return' being the only way to exit.

EditExit

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#*

WindowFont sets the font for the currently used window. Any further printing to this window will be in the specified font. IntuiFont# specifies a previously initialized intuifont object created using LoadFont.

WColour *Foreground Colour[,Background Colour]*

WColour 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 *Jammode*

WJam sets the text drawing mode of the currently used window. These drawing modes allow you to do inverted, complemented and other types of gaphics. The drawing modes can be OR'ed together to create a combination of them.

Jam1=0
This draws only the foreground colour and leavesthe background transparent. Eg For the letter O, any empty space (inside and outside the letter) will be transparent.

Jam2=1
This draws both the foreground and background to the window. Eg With the letter O again, the O will be drawn, but any clear area (inside and outside) will be drawn in the current background colour.

Complement=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.

Inversvid =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 will active the window specified by Window#.

Menus ***On/Off***

The Menus command 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#***

WPointer allows you to determine the mouse pointer imagery used in the currently used window. Shape# specifies an initialized shape object the pointer is to take it's appearance from, and must be of 2 bitplanes depth (4 colours).

WMove ***X,Y***

WMove will move the current window to screen position X,Y.

WSize ***Width,Height***

WSize will alter the width and height of the current window to the values specified by Width and Height.

WMouseX

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

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

EMouseX will return the horizontal position of the mouse pointer at the time the most recent window event occured. Window events are detected using the WaitEvent or Event commands.

EMouseY

EMouseY will return the vertical position of the mouse pointer at the time the most recent window event occured. Window events are detected using the WaitEvent or Event commands.

WCursX

WCursX returns the horizontal location of the text cursor of the currently used window. The text cursor position may be set using WLocate.

WCursY

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*

WLocate is 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 it's own text cursor position, therefore changing the text cursor position of one window will not affect any other window's text cursor position.

WindowX

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

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

WindowWidth returns the pixel width of the currently used window.

WindowHeight

WindowHeight returns the pixel height of the currently used window.

InnerWidth

InnerWidth returns the pixel width of the area inside the border of the currently used window.

InnerHeight

InnerHeight returns the pixel height of the area inside the border of the currently used window.

WTopOff

WTopOff returns the number of pixels between the top of the current window border and the inside of the window.

WLeftOff

WLeftOff returns the number of pixels between the left edge of the current window border and the inside of the window.

SizeLimits *Min Width,Min Height,Max Width,Max Height*

SizeLimits 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 *(Winodw#)*

RastPort 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*

PositionSuperBitMap is 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$,screentitle$*

WTitle is used to alter both the current window's title bar and it's screens title bar. Useful for displaying important stats such as program status etc.

CloseWindow *Window#*

CloseWindow has been added for convenience. Same as Free Window but a little more intuitive (added for those that have complained about such matters).

WPrintScroll

WPrintScroll will 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***

WBlit can be 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#[,srcx,srcy,destx,desty,wid,height]***

BitMaptoWindow will copy 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 inyerface.

EventCode

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.

EventQualifier

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]| [[Tag,Data]...]***

Similar to ScreenTags, WindowTags allows the advanced user to open a Blitz window with a list of OS Tags as descibed in documentation for the operating system prior to 2.0.

LoadFont ***IntuiFont#,Fontname.font$,Y Size [,style]***

LoadFont is 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.
The LoadFont command has been extended with an optional style parameter. The following constants may be combined:

```
#underlined=1
#bold=2
#italic=4
#extended=8 ;wider than normal
#colour=64 ;hmm use colour version I suppose
```

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.

The following is a table of the gadget flags and the gadget types which they are relevant to:

Bit#	Meaning	Text	String	Prop	Shape
0	Toggle On/Off	yes	no	no	yes
1	Relative to Right Side of Window	yes	yes	yes	yes
2	Relative to Bottom of Window	yes	yes	yes	yes
3	Size Relative to Width of Window	no	no	yes	no
4	Size Relative to Height of Window	no	no	yes	no
0	Box Select	yes	yes	yes	yes
6	Prop Gadget has Horizontal Movement	no	no	yes	no
7	Prop Gadget Has Vertical Movement	no	no	yes	no
8	No Border around Prop Gadget	no	no	yes	no
9	Mutually Exclusive	yes	yes	no	no
10	Attach to Window's Right Border	yes	yes	yes	yes
11	Attach to Window's Left Border	yes	yes	yes	yes
12	Attach to Window's Top Border	yes	yes	yes	yes
13	Attach to Window's Bottom Border	yes	yes	yes	yes
14	Use GimmeZeroZero Border	yes	yes	yes	yes

Note:

If Relative Right is set the gadgets X should be negative, as so 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 witdh as WindowWidth+GadgetWidth as it does height when relative size flags are set.

Mutually exclusive radio button type gadgets DO NOT require WorkBench 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.

If you create a text gadget which contains the '| ' character in the gadget's text, Blitz will recognize this as a 'cycling' gadget, using the '| ' character to separate the options - like this:

```
TextGadget 0,16,16,0,1," 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$***

The TextGadget 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.
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 gadget has been selected. Id may be any positive, non-zero number.
Text$ is the actual text you want the gadget to contain.

ButtonGroup ***Group***

ButtonGroup 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 specified 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 executed will belong to group 1.

SetGadgetStatus ***GadgetList#,Id,Value***

SetGadgetStatus is used to set a cycling text gadget to a particular value, once set ReDraw should be used to refresh the gadget to reflect it's new value.

GadgetPens Foreground Colour[,Background Colour]

GadgetPens 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 ***Jammode***

GadgetJam 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***

SelectMode is used to predefine how gadget rendering will show a gadget .selection, modes are 1 for box and 0 for inverse. Use prior to creation of gadgets.

ShapeGadget ***GadgetList#,X,Y,Flags,Id,Shape#[,Shape#]***

The ShapeGadget command allows you to create gadgets with graphic imagery. The Shape# parameter refers to a shape object containing the graphics you wish the gadget to contain.
The ShapeGadget 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***

StringGadget allows you 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 from 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.
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 gadget has been selected. Id may be any positive, non-zero number.
Maxlen refers to the maximum number of characters which may appear in this gadgets.
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)***

The Stringtext$ function allows you to determine the current contents of a string gadget. StringText$ will return a string of characters representing the string gadgets contents.

ActivateString *Window#,Id*

ActivateString 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 for the mouse before the keyboard.

ResetString *GadgetList#,Id*

ResetString allows you to 'reset' a string gadget. This will cause the string gadget's cursor position to be set to the leftmost position.

ClearString *GadgetList#,Id*

ClearString may be used to clear, or erase, the text in the specified string gadget. The cursor position will also be moved to the leftmost 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$*

SetString may be used to initialize 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 exeucte ReDraw to display the change.

PropGadget *GadgetList#,X,Y,Flags,Id,Width,Height*

The PropGadget command is 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. Proportional gadgets are commonly used for the 'R G B' 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 '.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 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 gadgets position, relative to the 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.
Id is a unique, non zero number which allows you to identify when the gadget is manipulated.
Proportional gadgets may be altered using the SetVProp and SetHProp commands, and read using the VPropPot, VPropBody, HPropPot and HPropBody functions.

SetHProp ***GadgetList#,Id,Pot,Body***

SetHProp is used to alter the horizontal slider qualities of a proportional gadget. Both Pot and Body should be in the range 0 through 1.
If SetHProp is executed while the specified gadget is already displayed, execution of 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***

SetVProp is used to alter the vertical slider qualities of a proportional gadget. Both Pot and Body should be in the range 0 through 1.
If SetVProp is executed while the specified gadget is already displayed, execution of the ReDraw command will be necessary to display the changes.

HPropPot ***(GadgetList#,Id)***

The HPropPot function allows you to determine the current 'pot' setting of a proportional gadget. HPropPot will return a number from 0 up to, but not including, 1, reflecting the gadgets current horizontal pot setting.

HPropBody ***(GadgetList#,Id)***

The HPropBody function allows you to determine the current 'body' setting of a proportional gadget. HPropBody will return a number from 0 up to, but not including, 1, reflecting the gadgets current horizontal body setting.

VPropPot ***(GadgetList#,Id)***

The VPropPot function allows you to determine the current 'pot' setting of a proportional gadget.
VPropPot will return a number from 0 up to, but not including, 1, reflecting the gadgets current vertical pot setting.

VPropBody ***(GadgetList#,Id)***

The VPropBody function allows you to determine the current 'body' setting of a proportional gadget.
VPropBody will return a number from 0 up to, but not including, 1, reflecting the gadgets current vertical body setting.

Redraw ***Window#,id***

ReDraw 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]| [Width,Height]*

Borders serves 2 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 it's border, Width referring to the left/right spacing, and Height to the above/below spacing.

BorderPens *Highlight Colour,Shadow Colour*

BorderPens allows you to control the colours used when gadget borders are created. Gadget borders may be created by the TextGadget, StringGadget and GadgetBorder commands.
HighLight Colour refers to the colour of the top and left edges of the border, while Shadow Colour refers to the right and bottom edges.
The default value for HighLight Colour is 1. The default value for Shadow Colour is 2.

GadgetBorder *X,Y,Width,Height*

The GadgetBorder command may be used to draw a rectangular border into the currently used window.
Proportional gadgets and shape gadgets do not have borders automatically created for them. The GadgetBorder 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. GadgetBorder will automatically insert sapces between the gadget and the border. The Borders command 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)*

GadgetStatus may be used to determine the status of the specified gadget. In the case of 'toggle' type gadget, 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)*

ButtonId may be used to determine which gadget within a group of button type gadgets is currently selected. The value returned will be the GadgetId of the button gadget currently selected.

Enable *GadgetList#,Id*

A gadget when disabled is covered by a "mesh" and can not be accessed by the user. The commands Enable and Disable allow the programmer to access this feature of Intuition.

Disable *GadgetList#,Id*

A gadget when disabled is covered by a "mesh" and can not be accessed by the user. The commands Enable and Disable allow the programmer to access this feature of Intuition.

Toggle *GadgetList#,Id [,On| Off]*

The Togggle command in the gadget library has been extended so it will actually toggle a gadgets status if the no On| Off parameter is missing.

R-25: MENU COMMANDS

Blitz supports many commands for the creation and use of Intuition menus.

Menus are created through the use of MenuList objects. Each menulist contains an entire set of menu titles, menu items and possibly sub menu items.

Menulists 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$*

MenuTitle is 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 leftmost menu. Menutitles 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$[,Shortcut$]*

MenuItem is used to create a text menu item. Menu items appear vertically below menu titles when the mouse is moved over a menu title with the right mouse button held down.
Flags affects the operation of the menu item.
A value of 0 creates a stand 'seleect' 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 be in 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 the topmost item. Menu items should be added in 'top down' order, with menu 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#***

ShapeItem is used to create a graphical menu item.
Shape# refers to a previously initialized 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,Subitemtext$[,Shortcut$]***

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 SubItem command.
Item specifies the menu item to attach the sub item to.
Subitem refers to the number of the sub menu item 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 character, specified using the Shortcut$ paramater.
All other parameters are identical to the MenuItem command.

ShapeSub ***MenuList#,Flags,Menu,Item,Subitem,Shape#***

ShapeSub allows you to create a graphic sub menu item. Shape# specifies a previously created shape object to be used as the sub item's grpahics.
All other parameters are identical to those in SubItem.

SetMenu ***MenuList#***

SetMenu is used to attach a menulist to the currently used window. Each window may have only one menulist attached to it.

MenuGap ***X Gap,Y Gap***

Executing MenuGap before creating any menu titles, items or sub items, allows you to control the layout of the menu.
X Gap refers to an amount, specified in pixles, to be inserted to the left and right of all menu items and sub menu items. Y Gap refers to an amount, again in pixels, to be inserted above and below all menu items and sub menu items.

SubItemOff ***X Offset,Y Offset***

SubItemOff allows you to control the relative position of the top of a list of sub menu items, in relation to their associated menu item.
Whenver 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***

The MenuState command allows you to turn menus, or sections of menus, on or off.
MenuState with just the MenuList# parameter may be 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***

MenuColour allows you to determine what colour any menu item or sub item text is rendered in. MenuColour should be executed before the approprate menu item commands.

MenuChecked ***(MenuList#,Menu,Item[,Subitem])***

The MenuChecked function 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, MenuChecked will return 'true' (-1). If not, MenuChecked will return 'false' (0).

R-26: GADTOOLS COMMANDS

GadTools are a new system of Gadgets added to the Amiga's operating system in version 2.0. They are improved in both looks and performance over the older standard Gadgets.

In order for certain GadTools gadgets to function 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 user holds down an arrow
AddIDCMP #MOUSEMOVE+#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 it's new value.

Use GTGetString and GTGetInteger functions to read the contents of the GadTools string gadgets after a #GadgetUp message.

GTadgetFlag		Value	
#_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 eg. "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()***

The ListView gadget enaables the 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$,flags,Options$***

GTMX 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 (user cannot interact with it) used to display numbers. See GTSetInteger to update the contents of this readonly "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 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 AttchGTList command is used to attach a set of GadTools gadgets to a Window after it has been opened.

GTTags ***Tag,Value [,Tag,Value...]***

The GTTags command 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=$80080000
#GTCB_Checked=#tag+4           ; State of checkbox
#GTLV_Top=#tag+5               ; Top visible item in listview
#GTLV_ReadOnly=#tag+7          ; Set TRUE if listview is ReadOnly
#GTMX_Active=#tag+10           ; Active one in mx gadget
#GTTX_Text=#tag+11             ; Text to display
#GTNM_Number=#tag+13           ; Number to display
#GTCY_Active=#tag+15           ; The active one in the cycle gad
#GTPA_Color=#tag+17            ; Palette color
#GTPA_ColorOffset=#tag+18      ; First color to use in palette
#GTSC_Top=#tag+21              ; Top visible in scroller
#GTSC_Total=#tag+22            ; Total in scroller area
#GTSC_Visible=#tag+23          ; Number visible in scroller
#GTSL_Level=#tag+40            ; Slider level
#GTSL_MaxLevelLen=#tag+41      ; Max length of printed level
#GTSL_LevelFormat=#tag+42      ;* Format string for level
#GTSL_LevelPlace=#tag+43       ;* Where level should be placed
#GTLV_Selected=#tag+54         ; Set ordinal number of selected
#GTMX_Spacing=#tag+61          ;* 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$="%2ld"
GTTags #GTSLLevelFormat,&f$,#GTSLMaxLevelLen,4
GTSlider 2,10,320,120,200,20,"GTSLIDER",2,0,10
```

GTGadPtr ***(GTList#,id)***

GTGadPtr returns the actual location of the specified GadTools gadget in memory.

GTBevelBox ***GTList#,x,y,w,h,flags***

GTBevelBox 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()]***

GTChangeList 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 reattache it with another call to GTChangeList this time using the List() parameter.

GTSetAttrs ***GTList#,id [,Tag,Value...]***

GTSetAttrs 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 GTInteger gadget.

GTGetAttrs ***(GTList#,id,Tag)***

A 3.0 specific command. See C= 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]***

GTToggle allows the programmer to set Boolean gadgets such as GTButton and GTCheckbox to a desired state.

GTStatus ***(GTList#,Id)***

GTStatus returns the status of and gadtools toggle gadgets, a value of 1 means the the gadget is selected, 0 deselected.

R-27: ASL LIBRARY COMMANDS

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 what ever size you want. I made it call the Blitz file requester if the program is running under 1.3 (isn't that nice!). There is a fix that patches the ReqTools file requester but that doesn't have the date field.
I couldn't get the Save-Only tag or the "Create Directory" option working maybe next upgrade.

**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 *(enable_flags)*

The ASL Font Requester is also pretty useful. 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 (just like programming in a high level language man!).

ASLScreenRequest *(enable_flags)*

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 (what are ya?).

R-28: AREXX CONTROL COMMANDS

ARexx allows communication between different Amiga applications allowing for some extensive and powerful control over applications by the programmer.

CreateMsgPort *("Name")*

CreateMsgPort is a general Function and not specific to ARexx.
CreateMsgPort 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)*

DeleteMsgPort 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")*

CreateRexxMsg() 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. i.e. 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***

DeleteRexxMsg 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 returned by CreateRexxMsg().

```
msg.l=CreateRexxMsg(Port,"rexx","HostName")
IF msg=0 THEN Error_Routine{}
DeleteRexxMsg msg
```

Again if you neglect to delete the RexxMsg structure Blitz will do this for you on exit of the program.

ClearRexxMsg ****rexxmsg***

ClearRexxMsg is used to delete and clear an ArgString from one ormore 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*** **)**

FillRexxMsg 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 acomplished 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 theLONG 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 args see #RXFUNC
SendRexxCommand msg,"",#RXFUNC| #RXFF_RESULT| 3
```

CreateArgString *("this is a string")*

CreateArgString() 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*

DeleteArgString is 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,"commandstring",#RXCOMM|#RXFF_RESULT*

SendRexxCommand is 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().

commandstring: 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 ARGO 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 ARGO 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 ARGO 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 and should 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 ARGO must contain an argstring Pointer to the command string. The RXFB_TOKEN flag specifies that the command line is to be tokenized 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_NOIO 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 ARGO 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_NOIO 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 ARGO 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 found 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 ARGO 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 tht 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 the command or function argument(in slot ARGO)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 tokenized 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 i.e. 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 Result1 and 2 are 0.

ReplyRexxMsg rexxmsg,0,0,"THE RETURNED MESSAGE"

GetRexxResult() ***Result.l=GetRexxResult(rexxmsg,ResultNum)***

GetRexxResult 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 a 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)***

GetRexxCommand 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.

Arguments are:

rexxmsg is a LONGWORD address of the RexxMsg structure as returned by RexxEvent()
ARGNum 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 ***(rexxmsg)***

GetResultString 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

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)***

RexxEvent is our Arexx Equivalent of EVENT(). It's purpose is to check the given Port to see if there is a message waiting there for us.
It should be called after 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):etc
```

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)***

IsRexxMsg 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 safely return as FALSE.

RexxError() ***ErrorString$=RexxError(ErrorCode)***

RexxError converts a numerical error code such as you would get from GetRexxResult(msg,2) into an understandable string error message. If the ErrorCode is not known to ARexx a string stating so is returned this ensures that this function will always succeed.

R-29: BREXX COMMANDS

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 your programs 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*

AbsMouse 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 *X Offset, Y Offset*

RelMouse allows you to move the mouse pointer a relative distance from it's 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*

MouseButton 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*

ClickButton is 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$*

Type 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#]***

Record 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#]***

PlayBack 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***

QuickPlay 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 it's 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

PlayWait may be used to halt program flow until a PlayBack of a tape has finished.

XStatus

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 either 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$***

SaveTape allows you to save a previously created tape object out to disk. This tape may later be reloaded using LoadTape.

LoadTape ***Tape#,Filename$***

LoadTape allows you to load a tape object previously saved with SaveTape for use with the PlayBack command.

TapeTrap *[Tape#]*

TapeTrap 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*

QuietTrap 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*

MacroKey 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*

FreeMacroKey causes a previously defined macro key to be removed so that a BRex tape is no longer attatched to it.

R-30: SERIAL PORT COMMANDS

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 on refer to any extra serial ports available.

OpenSerial *device$,unit#,baud,io_serflags*

OpenSerial is 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*

WriteSerial 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*

WriteSerialString is similar to WriteSerial but sends a complete string to the serial port.

ReadSerial *(unit#)*

ReadSerial returns the next byte waiting in the serial port's read buffer. If the buffer is empty it returns a -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#)*

ReadSerialString 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#***

The CloseSerial command will close 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***

SetSerialBuffer 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***

SetSerialLens 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#)***

SerialEvent is used when your program is handling events from more than 1 source, Windows, ARexx etc. This command is currently not implemented

ReadSerialMem ***Unit#,Address,Length***

ReadSerialMem will fill the given memory space with data from the given serial port.

WriteSerialMem ***Unit#,Address,Length***

WriteSerialMem send 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 paramaters. Use the HELP key for correct number and meaning of parameters with Blitz][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 syntatically incorrect.

Illegal End Procedure: The statement or function end is syntatically 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 only defined from the primary 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 syntatically 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: The 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.

Illgeal Absolute: The Absolute location specified must be defined and in range.

Illegal Displacement: The 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': Blitz][cannot find the library routine referred to by a Token, usually caused by the library not being included in DefLibs 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: The 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: Blitz 2 has a limit of 32K for any blocks of code such as IF..ENDIF blocks.

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 to Long: Blitz 2 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 types: The Exchange command can only swap two variables of the same type.

Can't Exchange NewTypes: The Exchange command can not 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: Blitz 2 cannot find the Resident file listed in the Options requester. Check the pathname.

Macro Based Errors

Macro Buffer Overflow: The Options requester in the Blitz 2 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 wellas somewhere 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 Blitz 2. 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, Blitz 2 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 paramater with the Dim statement.

Array not yet Dim'd: See Array not found.

Array not Dim'd: See Array not found.

Interupt Based Errors

End SetInt without SetInt: SetInt..SetInt is a block directive and both commands must be present.

SetInt without End SetInt: SetInt..SetInt is a block directive and both commands must be present.

Can't use Set/ClrInt 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 Set/ClrInt in Local Mode: Error handling can only be defined by the primary code.

End SetInt without SetInt: SetInt..SetInt is a block directive and both commands must be present.

SetInt without End SetInt: SetInt..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 the same name.

Illegal Label Name: Refer to the Programming in Blitz][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: The library is only available from within your code.

Illegal direct mode command: Direct mode is unable to execute the 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

Optimizer 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. Parameter details for each command are given in brackets and are also available via the Blitz 2 keyboard help system.

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.

The relative offsets from the library base and 68000 register parameters are included for the convenience of the assembler programmer. When using library calls an underscore character (_) should follow the token name.

An asterisk (*) preceding routine names specifies that the calls are private and should not be called from Blitz 2.

EXEC

```
-30 Supervisor(userFunction)(a5)
---- special patchable hooks to internal exec activity ---
 -36 *execPrivate1()()
 -42 *execPrivate2()()
 -48 *execPrivate3()()
 -54 *execPrivate4()()
 -60 *execPrivate5()()
 -66 *execPrivate6()()
--- module creation ---
 -72 InitCode(startClass,version)(d0/d1)
 -78 InitStruct(initTable,memory,size)(a1/a2,d0)
 -84 MakeLibrary(funcInit,structInit,libInit,dataSize,segList)(a0/a1/a2,d0/d1)
 -90 MakeFunctions(target,functionArray,funcDispBase)(a0/a1/a2)
 -96 FindResident(name)(a1)
-102 InitResident(resident,segList)(a1,d1)
--- diagnostics ---
-108 Alert(alertNum)(d7)
-114 Debug(flags)(d0)
--- interrupts ---
-120 Disable()()
-126 Enable()()
-132 Forbid()()
-138 Permit()()
-144 SetSR(newSR,mask)(d0/d1)
-150 SuperState()()
-156 UserState(sysStack)(d0)
-162 SetIntVector(intNumber,interrupt)(d0/a1)
-168 AddIntServer(intNumber,interrupt)(d0/a1)
-174 RemIntServer(intNumber,interrupt)(d0/a1)
-180 Cause(interrupt)(a1)
--- memory allocation ---
-186 Allocate(freeList,byteSize)(a0,d0)
-192 Deallocate(freeList,memoryBlock,byteSize)(a0/a1,d0)
-198 AllocMem(byteSize,requirements)(d0/d1)
-204 AllocAbs(byteSize,location)(d0/a1)
-210 FreeMem(memoryBlock,byteSize)(a1,d0)
-216 AvailMem(requirements)(d1)
-222 AllocEntry(entry)(a0)
```

-228 FreeEntry(entry)(a0)
--- lists ---
-234 Insert(list,node,pred)(a0/a1/a2)
-240 AddHead(list,node)(a0/a1)
-246 AddTail(list,node)(a0/a1)
-252 Remove(node)(a1)
-258 RemHead(list)(a0)
-264 RemTail(list)(a0)
-270 Enqueue(list,node)(a0/a1)
-276 FindName(list,name)(a0/a1)
--- tasks ---
-282 AddTask(task,initPC,finalPC)(a1/a2/a3)
-288 RemTask(task)(a1)
-294 FindTask(name)(a1)
-300 SetTaskPri(task,priority)(a1,d0)
-306 SetSignal(newSignals,signalSet)(d0/d1)
-312 SetExcept(newSignals,signalSet)(d0/d1)
-318 Wait(signalSet)(d0)
-324 Signal(task,signalSet)(a1,d0)
-330 AllocSignal(signalNum)(d0)
-336 FreeSignal(signalNum)(d0)
-342 AllocTrap(trapNum)(d0)
-348 FreeTrap(trapNum)(d0)
--- messages ---
-354 AddPort(port)(a1)
-360 RemPort(port)(a1)
-366 PutMsg(port,message)(a0/a1)
-372 GetMsg(port)(a0)
-378 ReplyMsg(message)(a1)
-384 WaitPort(port)(a0)
-390 FindPort(name)(a1)
--- libraries ---
-396 AddLibrary(library)(a1)
-402 RemLibrary(library)(a1)
-408 OldOpenLibrary(libName)(a1)
-414 CloseLibrary(library)(a1)
-420 SetFunction(library,funcOffset,newFunction)(a1,a0,d0)
-426 SumLibrary(library)(a1)
--- devices ---
-432 AddDevice(device)(a1)
-438 RemDevice(device)(a1)
-444 OpenDevice(devName,unit,ioRequest,flags)(a0,d0/a1,d1)
-450 CloseDevice(ioRequest)(a1)
-456 DoIO(ioRequest)(a1)
-462 SendIO(ioRequest)(a1)
-468 CheckIO(ioRequest)(a1)
-474 WaitIO(ioRequest)(a1)
-480 AbortIO(ioRequest)(a1)
--- resources ---
-486 AddResource(resource)(a1)
-492 RemResource(resource)(a1)
-498 OpenResource(resName)(a1)
--- private diagnostic support ---
-504 *execPrivate7()()
-510 *execPrivate8()()
-516 *execPrivate9()()
--- misc ---
-522 RawDoFmt(formatString,dataStream,putChProc,putChData)(a0/a1/a2/a3)
-528 GetCC()()
-534 TypeOfMem(address)(a1)
-540 Procure(semaport,bidMsg)(a0/a1)
-546 Vacate(semaport)(a0)
-552 OpenLibrary(libName,version)(a1,d0)
*** functions in Release 1.2 or higher ***
--- signal semaphores (note funny registers found in 1.2 or higher)---

-558 InitSemaphore(sigSem)(a0)
-564 ObtainSemaphore(sigSem)(a0)
-570 ReleaseSemaphore(sigSem)(a0)
-576 AttemptSemaphore(sigSem)(a0)
-582 ObtainSemaphoreList(sigSem)(a0)
-588 ReleaseSemaphoreList(sigSem)(a0)
-594 FindSemaphore(sigSem)(a1)
-600 AddSemaphore(sigSem)(a1)
-606 RemSemaphore(sigSem)(a1)
--- kickmem support ---
-612 SumKickData()()
--- more memory support ---
-618 AddMemList(size,attributes,pri,base,name)(d0/d1/d2/a0/a1)
-624 CopyMem(source,dest,size)(a0/a1,d0)
-630 CopyMemQuick(source,dest,size)(a0/a1,d0)
*** functions in Release 2.0 or higher ***
--- cache ---
-636 CacheClearU()()
-642 CacheClearE(address,length,caches)(a0,d0/d1)
-648 CacheControl(cacheBits,cacheMask)(d0/d1)
--- misc ---
-654 CreateIORequest(port,size)(a0,d0)
-660 DeleteIORequest(iorequest)(a0)
-666 CreateMsgPort()()
-672 DeleteMsgPort(port)(a0)
-678 ObtainSemaphoreShared(sigSem)(a0)
--- even more memory support ---
-684 AllocVec(byteSize,requirements)(d0/d1)
-690 FreeVec(memoryBlock)(a1)
-696 CreatePrivatePool(requirements,puddleSize,puddleThresh)(d0/d1/d2)
-702 DeletePrivatePool(poolHeader)(a0)
-708 AllocPooled(memSize,poolHeader)(d0/a0)
-714 FreePooled(memory,poolHeader)(a1,a0)
--- misc ---
-720 AttemptSemaphoreShared(sigSem)(a0)
-726 ColdReboot()()
-732 StackSwap(newStack)(a0)
--- task trees ---
-738 ChildFree(tid)(d0)
-744 ChildOrphan(tid)(d0)
-750 ChildStatus(tid)(d0)
-756 ChildWait(tid)(d0)
--- future expansion ---
-762 CachePreDMA(address,length,flags)(a0/a1,d1)
-768 CachePostDMA(address,length,flags)(a0/a1,d1)
-774 *execPrivate10()()
-780 *execPrivate11()()
-786 *execPrivate12()()
-792 *execPrivate13()()

DOS

-30 Open(name,accessMode)(d1/d2)
-36 Close(file)(d1)
-42 Read(file,buffer,length)(d1/d2/d3)
-48 Write(file,buffer,length)(d1/d2/d3)
-54 Input()()
-60 Output()()
-66 Seek(file,position,offset)(d1/d2/d3)
-72 DeleteFile(name)(d1)
-78 Rename(oldName,newName)(d1/d2)
-84 Lock(name,type)(d1/d2)
-90 UnLock(lock)(d1)
-96 DupLock(lock)(d1)
-102 Examine(lock,fileInfoBlock)(d1/d2)

-108 ExNext(lock,fileInfoBlock)(d1/d2)
-114 Info(lock,parameterBlock)(d1/d2)
-120 CreateDir(name)(d1)
-126 CurrentDir(lock)(d1)
-132 IoErr()()
-138 CreateProc(name,pri,segList,stackSize)(d1/d2/d3/d4)
-144 Exit(returnCode)(d1)
-150 LoadSeg(name)(d1)
-156 UnLoadSeg(seglist)(d1)
-162 *dosPrivate1()()
-168 *dosPrivate2()()
-174 DeviceProc(name)(d1)
-180 SetComment(name,comment)(d1/d2)
-186 SetProtection(name,protect)(d1/d2)
-192 DateStamp(date)(d1)
-198 Delay(timeout)(d1)
-204 WaitForChar(file,timeout)(d1/d2)
-210 ParentDir(lock)(d1)
-216 IsInteractive(file)(d1)
-222 Execute(string,file,file2)(d1/d2/d3)
*** functions in Release 2.0 or higher ***
---DOS Object creation/deletion---
-228 AllocDosObject(type,tags)(d1/d2)
-234 FreeDosObject(type,ptr)(d1/d2)
---Packet Level routines---
-240 DoPkt(port,action,arg1,arg2,arg3,arg4,arg5)(d1/d2/d3/d4/d5/d6/d7)
-246 SendPkt(dp,port,replyport)(d1/d2/d3)
-252 WaitPkt()()
-258 ReplyPkt(dp,res1,res2)(d1/d2/d3)
-264 AbortPkt(port,pkt)(d1/d2)
---Record Locking---
-270 LockRecord(fh,offset,length,mode,timeout)(d1/d2/d3/d4/d5)
-276 LockRecords(recArray,timeout)(d1/d2)
-282 UnLockRecord(fh,offset,length)(d1/d2/d3)
-288 UnLockRecords(recArray)(d1)
---Buffered File I/O---
-294 SelectInput(fh)(d1)
-300 SelectOutput(fh)(d1)
-306 FGetC(fh)(d1)
-312 FPutC(fh,ch)(d1/d2)
-318 UnGetC(fh,character)(d1/d2)
-324 FRead(fh,block,blocklen,number)(d1/d2/d3/d4)
-330 FWrite(fh,block,blocklen,number)(d1/d2/d3/d4)
-336 FGets(fh,buf,buflen)(d1/d2/d3)
-342 FPuts(fh,str)(d1/d2)
-348 VFWritef(fh,format,argarray)(d1/d2/d3)
-354 VFPrintf(fh,format,argarray)(d1/d2/d3)
-360 Flush(fh)(d1)
-366 SetVBuf(fh,buff,type,size)(d1/d2/d3/d4)
---DOS Object Management---
-372 DupLockFromFH(fh)(d1)
-378 OpenFromLock(lock)(d1)
-384 ParentOfFH(fh)(d1)
-390 ExamineFH(fh,fib)(d1/d2)
-396 SetFileDate(name,date)(d1/d2)
-402 NameFromLock(lock,buffer,len)(d1/d2/d3)
-408 NameFromFH(fh,buffer,len)(d1/d2/d3)
-414 SplitName(name,seperator,buf,oldpos,size)(d1/d2/d3/d4/d5)
-420 SameLock(lock1,lock2)(d1/d2)
-426 SetMode(fh,mode)(d1/d2)
-432 ExAll(lock,buffer,size,data,control)(d1/d2/d3/d4/d5)
-438 ReadLink(port,lock,path,buffer,size)(d1/d2/d3/d4/d5)
-444 MakeLink(name,dest,soft)(d1/d2/d3)
-450 ChangeMode(type,fh,newmode)(d1/d2/d3)
-456 SetFileSize(fh,pos,mode)(d1/d2/d3)

---Error Handling---
-462 SetIoErr(result)(d1)
-468 Fault(code,header,buffer,len)(d1/d2/d3/d4)
-474 PrintFault(code,header)(d1/d2)
-480 ErrorReport(code,type,arg1,device)(d1/d2/d3/d4)
-486 RESERVED
---Process Management---
-492 Cli()()
-498 CreateNewProc(tags)(d1)
-504 RunCommand(seg,stack,paramptr,paramlen)(d1/d2/d3/d4)
-510 GetConsoleTask()()
-516 SetConsoleTask(task)(d1)
-522 GetFileSysTask()()
-528 SetFileSysTask(task)(d1)
-534 GetArgStr()()
-540 SetArgStr(string)(d1)
-546 FindCliProc(num)(d1)
-552 MaxCli()()
-558 SetCurrentDirName(name)(d1)
-564 GetCurrentDirName(buf,len)(d1/d2)
-570 SetProgramName(name)(d1)
-576 GetProgramName(buf,len)(d1/d2)
-582 SetPrompt(name)(d1)
-588 GetPrompt(buf,len)(d1/d2)
-594 SetProgramDir(lock)(d1)
-600 GetProgramDir()()
---Device List Management---
-606 SystemTagList(command,tags)(d1/d2)
-612 AssignLock(name,lock)(d1/d2)
-618 AssignLate(name,path)(d1/d2)
-624 AssignPath(name,path)(d1/d2)
-630 AssignAdd(name,lock)(d1/d2)
-636 RemAssignList(name,lock)(d1/d2)
-642 GetDeviceProc(name,dp)(d1/d2)
-648 FreeDeviceProc(dp)(d1)
-654 LockDosList(flags)(d1)
-660 UnLockDosList(flags)(d1)
-666 AttemptLockDosList(flags)(d1)
-672 RemDosEntry(dlist)(d1)
-678 AddDosEntry(dlist)(d1)
-684 FindDosEntry(dlist,name,flags)(d1/d2/d3)
-690 NextDosEntry(dlist,flags)(d1/d2)
-696 MakeDosEntry(name,type)(d1/d2)
-702 FreeDosEntry(dlist)(d1)
-708 IsFileSystem(name)(d1)
---Handler Interface---
-714 Format(filesystem,volumename,dostype)(d1/d2/d3)
-720 Relabel(drive,newname)(d1/d2)
-726 Inhibit(name,onoff)(d1/d2)
-732 AddBuffers(name,number)(d1/d2)
---Date, Time Routines---
-738 CompareDates(date1,date2)(d1/d2)
-744 DateToStr(datetime)(d1)
-750 StrToDate(datetime)(d1)
---Image Management---
-756 InternalLoadSeg(fh,table,funcarray,stack)(d0/a0/a1/a2)
-762 InternalUnLoadSeg(seglist,freefunc)(d1/a1)
-768 NewLoadSeg(file,tags)(d1/d2)
-774 AddSegment(name,seg,system)(d1/d2/d3)
-780 FindSegment(name,seg,system)(d1/d2/d3)
-786 RemSegment(seg)(d1)
---Command Support----
-792 CheckSignal(mask)(d1)
-798 ReadArgs(template,array,args)(d1/d2/d3)
-804 FindArg(keyword,template)(d1/d2)

-810 ReadItem(name,maxchars,cSource)(d1/d2/d3)
-816 StrToLong(string,value)(d1/d2)
-822 MatchFirst(pat,anchor)(d1/d2)
-828 MatchNext(anchor)(d1)
-834 MatchEnd(anchor)(d1)
-840 ParsePattern(pat,buf,buflen)(d1/d2/d3)
-846 MatchPattern(pat,str)(d1/d2)
-852 * Not currently implemented.
-858 FreeArgs(args)(d1)
-864 *--- (1 function slot reserved here) ---
-870 FilePart(path)(d1)
-876 PathPart(path)(d1)
-882 AddPart(dirname,filename,size)(d1/d2/d3)
---Notification---
-888 StartNotify(notify)(d1)
-894 EndNotify(notify)(d1)
---Environment Variable functions---
-900 SetVar(name,buffer,size,flags)(d1/d2/d3/d4)
-906 GetVar(name,buffer,size,flags)(d1/d2/d3/d4)
-912 DeleteVar(name,flags)(d1/d2)
-918 FindVar(name,type)(d1/d2)
-924 *dosPrivate4()()
-930 CliInitNewcli(dp)(a0)
-936 CliInitRun(dp)(a0)
-942 WriteChars(buf,buflen)(d1/d2)
-948 PutStr(str)(d1)
-954 VPrintf(format,argarray)(d1/d2)
-960 *--- (1 function slot reserved here) ---
-966 ParsePatternNoCase(pat,buf,buflen)(d1/d2/d3)
-972 MatchPatternNoCase(pat,str)(d1/d2)
-978 dosPrivate5()()
-984 SameDevice(lock1,lock2)(d1/d2)

GRAPHICS

-30 BltBitMap
(srcBitMap,xSrc,ySrc,destBitMap,xDest,yDest,xSize,ySize,minterm,mask,tempA)
(a0,d0/d1/a1,d2/d3/d4/d5/d6/d7/a2)
-36
BltTemplate(source,xSrc,srcMod,destRP,xDest,yDest,xSize,ySize)(a0,d0/d1/a1,d2/d3/d4/d5)
--- Text routines ---
-42 ClearEOL(rp)(a1)
-48 ClearScreen(rp)(a1)
-54 TextLength(rp,string,count)(a1,a0,d0)
-60 Text(rp,string,count)(a1,a0,d0)
-66 SetFont(rp,textFont)(a1,a0)
-72 OpenFont(textAttr)(a0)
-78 CloseFont(textFont)(a1)
-84 AskSoftStyle(rp)(a1)
-90 SetSoftStyle(rp,style,enable)(a1,d0/d1)
--- Gels routines ---
-96 AddBob(bob,rp)(a0/a1)
-102 AddVSprite(vSprite,rp)(a0/a1)
-108 DoCollision(rp)(a1)
-114 DrawGList(rp,vp)(a1,a0)
-120 InitGels(head,tail,gelsInfo)(a0/a1/a2)
-126 InitMasks(vSprite)(a0)
-132 RemIBob(bob,rp,vp)(a0/a1/a2)
-138 RemVSprite(vSprite)(a0)
-144 SetCollision(num,routine,gelsInfo)(d0/a0/a1)
-150 SortGList(rp)(a1)
-156 AddAnimOb(anOb,anKey,rp)(a0/a1/a2)
-162 Animate(anKey,rp)(a0/a1)
-168 GetGBuffers(anOb,rp,flag)(a0/a1,d0)

-174 InitGMasks(anOb)(a0)
--- General graphics routines ---
-180 DrawEllipse(rp,xCenter,yCenter,a,b)(a1,d0/d1/d2/d3)
-186 AreaEllipse(rp,xCenter,yCenter,a,b)(a1,d0/d1/d2/d3)
-192 LoadRGB4(vp,colors,count)(a0/a1,d0)
-198 InitRastPort(rp)(a1)
-204 InitVPort(vp)(a0)
-210 MrgCop(view)(a1)
-216 MakeVPort(view,vp)(a0/a1)
-222 LoadView(view)(a1)
-228 WaitBlit()()
-234 SetRast(rp,pen)(a1,d0)
-240 Move(rp,x,y)(a1,d0/d1)
-246 Draw(rp,x,y)(a1,d0/d1)
-252 AreaMove(rp,x,y)(a1,d0/d1)
-258 AreaDraw(rp,x,y)(a1,d0/d1)
-264 AreaEnd(rp)(a1)
-270 WaitTOF()()
-276 QBlit(blit)(a1)
-282 InitArea(areaInfo,vectorBuffer,maxVectors)(a0/a1,d0)
-288 SetRGB4(vp,index,red,green,blue)(a0,d0/d1/d2/d3)
-294 QBSBlit(blit)(a1)
-300 BltClear(memBlock,byteCount,flags)(a1,d0/d1)
-306 RectFill(rp,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3)
-312 BltPattern(rp,mask,xMin,yMin,xMax,yMax,maskBPR)(a1,a0,d0/d1/d2/d3/d4)
-318 ReadPixel(rp,x,y)(a1,d0/d1)
-324 WritePixel(rp,x,y)(a1,d0/d1)
-330 Flood(rp,mode,x,y)(a1,d2,d0/d1)
-336 PolyDraw(rp,count,polyTable)(a1,d0/a0)
-342 SetAPen(rp,pen)(a1,d0)
-348 SetBPen(rp,pen)(a1,d0)
-354 SetDrMd(rp,drawMode)(a1,d0)
-360 InitView(view)(a1)
-366 CBump(copList)(a1)
-372 CMove(copList,destination,data)(a1,d0/d1)
-378 CWait(copList,v,h)(a1,d0/d1)
-384 VBeamPos()()
-390 InitBitMap(bitMap,depth,width,height)(a0,d0/d1/d2)
-396 ScrollRaster(rp,dx,dy,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3/d4/d5)
-402 WaitBOVP(vp)(a0)
-408 GetSprite(sprite,num)(a0,d0)
-414 FreeSprite(num)(d0)
-420 ChangeSprite(vp,sprite,newData)(a0/a1/a2)
-426 MoveSprite(vp,sprite,x,y)(a0/a1,d0/d1)
-432 LockLayerRom(layer)(a5)
-438 UnlockLayerRom(layer)(a5)
-444 SyncSBitMap(layer)(a0)
-450 CopySBitMap(layer)(a0)
-456 OwnBlitter()()
-462 DisownBlitter()()
-468 InitTmpRas(tmpRas,buffer,size)(a0/a1,d0)
-474 AskFont(rp,textAttr)(a1,a0)
-480 AddFont(textFont)(a1)
-486 RemFont(textFont)(a1)
-492 AllocRaster(width,height)(d0/d1)
-498 FreeRaster(p,width,height)(a0,d0/d1)
-504 AndRectRegion(region,rectangle)(a0/a1)
-510 OrRectRegion(region,rectangle)(a0/a1)
-516 NewRegion()()
-522 ClearRectRegion(region,rectangle)(a0/a1)
-528 ClearRegion(region)(a0)
-534 DisposeRegion(region)(a0)
-540 FreeVPortCopLists(vp)(a0)
-546 FreeCopList(copList)(a0)

-552
ClipBlit(srcRP,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,minterm)(a0,d0/d1/a1,d2/d3/d4/d5/d6)
-558 XorRectRegion(region,rectangle)(a0/a1)
-564 FreeCprList(cprList)(a0)
-570 GetColorMap(entries)(d0)
-576 FreeColorMap(colorMap)(a0)
-582 GetRGB4(colorMap,entry)(a0,d0)
-588 ScrollVPort(vp)(a0)
-594 UCopperListInit(uCopList,n)(a0,d0)
-600 FreeGBuffers(anOb,rp,flag)(a0/a1,d0)
-606 BltBitMapRastPort(srcBM,x,y,destRP,x,y,WId,Height,minterm)(a0,d0/d1/a1,d2/d3/d4/d5/d6)
-612 OrRegionRegion(srcRegion,destRegion)(a0/a1)
-618 XorRegionRegion(srcRegion,destRegion)(a0/a1)
-624 AndRegionRegion(srcRegion,destRegion)(a0/a1)
-630 SetRGB4CM(colorMap,index,red,green,blue)(a0,d0/d1/d2/d3)
-636 BltMaskBitMapRastPort
(srcBM,x,y,destRP,x,y,Wid,High,mterm,Mask)(a0,d0/d1/a1,d2/d3/d4/d5/d6/a2)
-642 RESERVED
-648 RESERVED
-654 AttemptLockLayerRom(layer)(a5)
*** functions in Release 2.0 or higher ***
-660 GfxNew(gfxNodeType)(d0)
-666 GfxFree(gfxNodePtr)(a0)
-672 GfxAssociate(associateNode,gfxNodePtr)(a0/a1)
-678 BitMapScale(bitScaleArgs)(a0)
-684 ScalerDiv(factor,numerator,denominator)(d0/d1/d2)
-690 TextFit
(rp,string,strLen,textExtent,constrainingExtent,strDirection,constrainingBitWidth,constrainingBitHeight)(a1,a0,d0/a2)

INTUITION

-30 OpenIntuition()()
-36 Intuition(iEvent)(a0)
-42 AddGadget(window,gadget,position)(a0/a1,d0)
-48 ClearDMRequest(window)(a0)
-54 ClearMenuStrip(window)(a0)
-60 ClearPointer(window)(a0)
-66 CloseScreen(screen)(a0)
-72 CloseWindow(window)(a0)
-78 CloseWorkBench()()
-84 CurrentTime(seconds,micros)(a0/a1)
-90 DisplayAlert(alertNumber,string,height)(d0/a0,d1)
-96 DisplayBeep(screen)(a0)
-102 DoubleClick(sSeconds,sMicros,cSeconds,cMicros)(d0/d1/d2/d3)
-108 DrawBorder(rp,border,leftOffset,topOffset)(a0/a1,d0/d1)
-114 DrawImage(rp,image,leftOffset,topOffset)(a0/a1,d0/d1)
-120 EndRequest(requester,window)(a0/a1)
-126 GetDefPrefs(preferences,size)(a0,d0)
-132 GetPrefs(preferences,size)(a0,d0)
-138 InitRequester(requester)(a0)
-144 ItemAddress(menuStrip,menuNumber)(a0,d0)
-150 ModifyIDCMP(window,flags)(a0,d0)
-156 ModifyProp
(gadget,window,requester,flags,horizPot,vertPot,horizBody,vertBody)(a0/a1/a2,d0/d1/d2/d3/d4)
-162 MoveScreen(screen,dx,dy)(a0,d0/d1)
-168 MoveWindow(window,dx,dy)(a0,d0/d1)
-174 OffGadget(gadget,window,requester)(a0/a1/a2)
-180 OffMenu(window,menuNumber)(a0,d0)
-186 OnGadget(gadget,window,requester)(a0/a1/a2)
-192 OnMenu(window,menuNumber)(a0,d0)
-198 OpenScreen(newScreen)(a0)
-204 OpenWindow(newWindow)(a0)
-210 OpenWorkBench()()

-216 PrintIText(rp,iText,left,top)(a0/a1,d0/d1)
-222 RefreshGadgets(gadgets,window,requester)(a0/a1/a2)
-228 RemoveGadget(window,gadget)(a0/a1)
-234 ReportMouse(flag,window)(d0/a0)
-240 Request(requester,window)(a0/a1)
-246 ScreenToBack(screen)(a0)
-252 ScreenToFront(screen)(a0)
-258 SetDMRequest(window,requester)(a0/a1)
-264 SetMenuStrip(window,menu)(a0/a1)
-270 SetPointer(window,pointer,height,width,xOffset,yOffset)(a0/a1,d0/d1/d2/d3)
-276 SetWindowTitles(window,windowTitle,screenTitle)(a0/a1/a2)
-282 ShowTitle(screen,showIt)(a0,d0)
-288 SizeWindow(window,dx,dy)(a0,d0/d1)
-294 ViewAddress()()
-300 ViewPortAddress(window)(a0)
-306 WindowToBack(window)(a0)
-312 WindowToFront(window)(a0)
-318 WindowLimits(window,widthMin,heightMin,widthMax,heightMax)(a0,d0/d1/d2/d3)
-324 SetPrefs(preferences,size,inform)(a0,d0/d1)
-330 IntuiTextLength(iText)(a0)
-336 WBenchToBack()()
-342 WBenchToFront()()
-348
AutoRequest(window,body,posText,negText,pFlag,nFlag,width,height)(a0/a1/a2/a3,d0/d1/d2/d3)
-354 BeginRefresh(window)(a0)
-360 BuildSysRequest(window,body,posText,negText,flags,width,height)(a0/a1/a2/a3,d0/d1/d2)
-366 EndRefresh(window,complete)(a0,d0)
-372 FreeSysRequest(window)(a0)
-378 MakeScreen(screen)(a0)
-384 RemakeDisplay()()
-390 RethinkDisplay()()
-396 AllocRemember(rememberKey,size,flags)(a0,d0/d1)
-402 AlohaWorkbench(wbport)(a0)
-408 FreeRemember(rememberKey,reallyForget)(a0,d0)
-414 LockIBase(dontknow)(d0)
-420 UnlockIBase(ibLock)(a0)
*** functions in Release 1.2 or higher ***
-426 GetScreenData(buffer,size,type,screen)(a0,d0/d1/a1)
-432 RefreshGList(gadgets,window,requester,numGad)(a0/a1/a2,d0)
-438 AddGList(window,gadget,position,numGad,requester)(a0/a1,d0/d1/a2)
-444 RemoveGList(remPtr,gadget,numGad)(a0/a1,d0)
-450 ActivateWindow(window)(a0)
-456 RefreshWindowFrame(window)(a0)
-462 ActivateGadget(gadgets,window,requester)(a0/a1/a2)
-468 NewModifyProp
(gadget,window,requester,flags,horizPot,vertPot,horizBody,vertBody,numGad)
(a0/a1/a2,d0/d1/d2/d3/d4/d5)
*** functions in Release 2.0 or higher ***
-474 QueryOverscan(displayID,rect,oScanType)(a0/a1,d0)
-480 MoveWindowInFrontOf(window,behindWindow)(a0/a1)
-486 ChangeWindowBox(window,left,top,width,height)(a0,d0/d1/d2/d3)
-492 SetEditHook(hook)(a0)
-498 SetMouseQueue(window,queueLength)(a0,d0)
-504 ZipWindow(window)(a0)
--- public screens ---
-510 LockPubScreen(name)(a0)
-516 UnlockPubScreen(name,screen)(a0/a1)
-522 LockPubScreenList()()
-528 UnlockPubScreenList()()
-534 NextPubScreen(screen,namebuf)(a0/a1)
-540 SetDefaultPubScreen(name)(a0)
-546 SetPubScreenModes(modes)(d0)
-552 PubScreenStatus(screen,statusFlags)(a0,d0)
-558 ObtainGIRPort(gInfo)(a0)
-564 ReleaseGIRPort(rp)(a0)

-570 GadgetMouse(gadget,gInfo,mousePoint)(a0/a1/a2)
-576 *intuitionPrivate1()()
-582 GetDefaultPubScreen(nameBuffer)(a0)
-588 EasyRequestArgs(window,easyStruct,idcmpPtr,args)(a0/a1/a2/a3)
-594 BuildEasyRequestArgs(window,easyStruct,idcmp,args)(a0/a1,d0/a3)
-600 SysReqHandler(window,idcmpPtr,waitInput)(a0/a1,d0)
-606 OpenWindowTagList(newWindow,tagList)(a0/a1)
-612 OpenScreenTagList(newScreen,tagList)(a0/a1)
---new Image functions---
-618 DrawImageState(rp,image,leftOffset,topOffset,state,drawInfo)(a0/a1,d0/d1/d2/a2)
-624 PointInImage(point,image)(d0/a0)
-630 EraseImage(rp,image,leftOffset,topOffset)(a0/a1,d0/d1)
-636 NewObjectA(classPtr,classID,tagList)(a0/a1/a2)
-642 DisposeObject(object)(a0)
-648 SetAttrsA(object,tagList)(a0/a1)
-654 GetAttr(attrID,object,storagePtr)(d0/a0/a1)
---special set attribute call for gadgets---
-660 SetGadgetAttrsA(gadget,window,requester,tagList)(a0/a1/a2/a3)
-666 NextObject(objectPtrPtr)(a0)
-672 *intuitionPrivate2()()
-678 MakeClass(classID,superClassID,superClassPtr,instanceSize,flags)(a0/a1/a2,d0/d1)
-684 AddClass(classPtr)(a0)
-690 GetScreenDrawInfo(screen)(a0)
-696 FreeScreenDrawInfo(screen,drawInfo)(a0/a1)
-702 ResetMenuStrip(window,menu)(a0/a1)
-708 RemoveClass(classPtr)(a0)
-714 FreeClass(classPtr)(a0)
-720 *intuitionPrivate3()()
-726 *intuitionPrivate4()()

DISKFONT

-30 OpenDiskFont(textAttr)(a0)
-36 AvailFonts(buffer,bufBytes,flags)(a0,d0/d1)
*** functions in Release 1.2 or higher ***
-42 NewFontContents(fontsLock,fontName)(a0/a1)
-48 DisposeFontContents(fontContentsHeader)(a1)
*** functions in Release 2.0 or higher ***
-54 NewScaledDiskFont(sourceFont,destTextAttr)(a0/a1)

APPENDIX 3: AMIGA HARDWARE REGISTERS

The following are a list of memory locations where direct access to the Agnus, Denise and Paula chips is possible. It is illegal to access any of these registers if you wish your program to behave correctly in the Amiga environment. However in BlitzMode most of these registers may be accessed taking into consideration the accompanying documentation.

An * next to any description states that the option is available only with the new ECS (Enhanced Chip Set). Also note that any reference to memory pointers MUST point to chip mem as the Amiga Chip Set is NOT capable of accessing FAST mem. This includes BitPlane data, copper lists, Sprite Data, Sound DATA etc. etc.

BitPlane & Display Control

The Amiga has great flexibility in displaying graphics at different resolutions and positions on the monitor. The hardware registers associated with the display are nearly always loaded by the copper and not with the 68000 processor.

```
#BPLCON0=$100
#BPLCON1=$102
#BPLCON2=$104
#BPLCON3=$106      ;(ECS only)
#BPLCON4=$10c      ;(AGA only)
```

BIT#	BPLCON0	BPLCON1	BPLCON2	BPLCON3	BPLCON4
15	HIRES			COLBANK2	BPLAM7
14	BPU2		ZDBPSEL2	COLBANK1	BPLAM6
13	BPU1		ZDBPSEL1	COLBANK0	BPLAM5
12	BPU0		ZDBPSEL0	PF2OF2	BPLAM4
11	HAM		ZDPEN	PF2OF1	BPLAM3
10	DBLPF		ZDCTEN	PF2OF0	BPLAM2
09	COLOR		KILLEHB	LOCT	BPLAM1
08	GAUD		RDRAM=0		BPLAM0
07		PF2H3	SOGEN	SPRES1	ESPRM7
06	*SHRES	PF2H2	PF2PRI	SPRES0	ESPRM6
05	*BPLHWRM	PF2H1	PF2P2	BRDRBLNK	ESPRM5
04	*SPRHWRM	PF2H0	PF2P1	BRDRTRAN	ESPRM4
03	LPEN	PF1H3	PF2P0		OSPRM7
02	LACE	PF1H2	PF1P2	ZDCLCKEN	OSPRM6
01	ERSY	PF1H1	PF1P1	BRDSPRT	OSPRM5
00		PF1H0	PF1P0	EXTBLKEN	OSPRM4

BPUn = number of bitplanes
PFnHn = playfield horizontal positioning
ZD... = genlock enable bits
PFnPn = Playfield priorities
COLBANKn = active color bank in AGA
PF2OFn = color offset for playfield 2 in dpf mode
LOCT = hi/lo nibble select for 24 bit color access
SPRESn = Sprite resolution
BRD... = Border settings
BPLAMx = xor mask for bitplane fetch
ESPRMn = color offset for even sprites

```
#BPL0PTH= $E0   ;BitPlane Pointer 0 High Word
#BPL0PTL= $E2   ;BitPlane Pointer 0 Low Word
#BPL1PTH= $E4
#BPL1PTL= $E6
#BPL2PTH= $E8
#BPL2PTL= $EA
#BPL3PTH= $EC
#BPL3PTL= $EE
#BPL4PTH= $F0
#BPL4PTL= $F2
#BPL5PTH= $F4
#BPL5PTL= $F6
```

Each pair of registers contain an 18 bit pointer to the address of BitPlanex data in chip memory. They MUST be reset every frame usually by the copper.

```
#BPL1MOD=$108  ;Bitplane Modulo for Odd Planes
#BPL2MOD=$10A  ;Bitplane Modulo for EvenPlanes
```

At the end of each display line, the BPLxMODs are added to the the BitPLane Pointers so they point to the address of the next line.

```
#DIWSTOP=$090  ; display window stop
#DIWSTRT=$08E  ; display window start
```

These two registers control the display window size and position. The following bits are assigned

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

For DIWSTRT V8=0 & H8=0 restricting it to the upper left of the screen. For DIWSTOP V8=1 & H8=1 restricting it to the lower right of the screen.

```
#DDFSTOP=$094        ; data fetch stop
#DDFSTRT= $092       ; data fetch start
```

The two display data fetch registers control when and how many words are fetched from the bitplane for each line of display.

Typical values are as follows:

lores 320 pixels, DDFSTRT & DDFSTOP = $38 & $D0
hires 640 pixels, DDFSTRT & DDFSTOP = $3C & $d4

If smooth scrolling is enabled DDFSTRT should be 2 less than above.

```
#BPL1DAT $110  ; BitPlane Data parallel to serial converters
#BPL2DAT $112
#BPL3DAT $114
#BPL4DAT $116
#BPL5DAT $118
#BPL6DAT $11A
```

These 6 registers receive the DMA data fetched by the BitPlane engine, and output it serially to the Amiga DACS, triggered by writing to BPL1DAT. Not intended for programmer access.

The Copper

The Copper is found on the Agnus chip, it's main job is to 'poke' values into the hardware registers in sync with the video beam. The main registers it updates are BitPlane ptrs, Sprites and other control words that HAVE to be reset every frame. It's also used to split the screen vertically as it is capable of waiting for certain video beam positions before writing data. Its also capable of waiting for the blitter to finish as well as skipping instructions if beam position is equal to certain values.

#COP1LCH=$080
#COP1LCL=$082

#COP2LCH=$084
#COP2LCL=$086

Each pair of registers contain an 18 bit pointer to the address of a Copper List in chip mem. The Copper will automatically jump to the address in COP1 at the beginning of the frame and is able to jump to COP2 if the following strobe is written to.

#COPJMP1=$88
#COPJMP2=$8A

When written to these addresses cause the copper to jump to the locations held in COP1LC & COP2LC. The Copper can write to these registers itself causing its own indirect jump.

#COPCON=$2E

By setting bit 1 of this register the copper is allowed to access the blitter hardware.

The copper fetches two words for each instruction from its current copper list. The three instructions it can perform and their relevant bits are as follows:

Bit#	MOVE		WAIT UNTIL		SKIP IF	
15	x	RD15	VP7	BFD	VP7	BFD
14	x	RD14	VP6	VE6	VP6	VE6
13	x	RD13	VP5	VE5	VP5	VE5
12	x	RD12	VP4	VE4	VP4	VE4
11	x	RD11	VP3	VE3	VP3	VE3
10	x	RD10	VP2	VE2	VP2	VE2
09	x	RD09	VP1	VE1	VP1	VE1
08	DA8	RD08	VP0	VE0	VP0	VE0
07	DA7	RD07	HP8	HE8	HP8	HE8
06	DA6	RD06	HP7	HE7	HP7	HE7
05	DA5	RD05	HP6	HE6	HP6	HE6
04	DA4	RD04	HP5	HE5	HP5	HE5
03	DA3	RD03	HP4	HE4	HP4	HE4
02	DA2	RD02	HP3	HE3	HP3	HE3
01	DA1	RD01	HP2	HE2	HP2	HE2
00	0	RD00	1	0	1	1

The MOVE instruction shifts the value held in RD15-0 to the destination address calculated by $DFF000 +DA8-1.

The WAIT UNTIL instruction places the copper in a wait state until the video beam position is past HP,VP (xy coordinates). The Copper first logical ANDS (masks) the video beam with HE,VE before doing the comparison. If BFD is set then the blitter must also be finished before the copper will exit its wait state.

The SKIP IF instruction is similar to the WAIT UNTIL instruction but instead of placing the copper in a wait state if the video beam position fails the comparison test it skips the next MOVE instruction.

Colour Registers

The following 32 color registers can each represent one of 4096 colors.

```
#COLOR00=$180  #COLOR08=$190   #COLOR16=$1A0  #COLOR24=$1B0
#COLOR01=$182  #COLOR09=$192   #COLOR17=$1A2  #COLOR25=$1B2
#COLOR02=$184  #COLOR10=$194   #COLOR18=$1A4  #COLOR26=$1B4
#COLOR03=$186  #COLOR11=$196   #COLOR19=$1A6  #COLOR27=$1B6
#COLOR04=$188  #COLOR12=$198   #COLOR20=$1A8  #COLOR28=$1B8
#COLOR05=$18A  #COLOR13=$19A   #COLOR21=$1AA  #COLOR29=$1BA
#COLOR06=$18C  #COLOR14=$19C   #COLOR22=$1AC  # COLOR30=$1BC
#COLOR07=$18E  #COLOR15=$19E   #COLOR23=$1AE  #COLOR31=$1BE
```

The bit usage for each of the 32 colors is:

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	x	x	x	x	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

This represents a combination of 16 shades of red, green and blue.

Blitter Control

The Blitter is located on the Agnus, it's main function is to move blocks of data around chip mem. It has 3 input channels A,B & C and 1 output channel D. A simple block move would use 1 input channel and the 1 output channel, taking 4 clock ticks per cycle. A complex move such as a moving a shape to a destination with a cookie cut would use all 3 input channels and the output channel taking 8 clock ticks per cycle.

The main parameters of the blitter include the width and height of the block to be moved (width is in multiples of words), a start address for each channel, a modulo for each channel that is added to there address at the end of each line so they point to the next line, a logic function that specifies which input channels data will be sent to the destination channel.

The following is a table to work out the logic function (known as the minterm) for a blitter operation.

A	B	C	D
0	0	0	LF0
0	0	1	LF1
0	1	0	LF2
0	1	1	LF3
1	0	0	LF4
1	0	1	LF5
1	1	0	LF6
1	1	1	LF7

If the Blitter is set up so that channel A points to the cookie, B points to the shape to be copied and C&D point to the destination bitplane (such as how Blitz 2 uses the blitter) we would specify the following conditions:

When A is 1 then make D=B
When A is 0 then make D=C

Using the above table we calculate the values of LF0-LF7 when these two conditions are met. The top line has A=0 so LF0 becomes the value in the C column which is a 0. A is 0 in the first 4 rows so LF0-LF3 all reflect the bits in the C column (0101) and A=1 in the lower 4 rows so LF4-LF7 reflect the bits in the B column (0011).

This generates a minterm LF0-LF7 of %10101100 or in hex $AC.

Note: read the values of LF7 to LF0 from bottom to top to calculate the correct hexadecimal minterm.

```
#BLTAPTH= $50
#BLTAPTL= $52

#BLTBPTH= $4C
#BLTBPTL= $4E

#BLTCPTH= $48
#BLTCPTL= $4A

#BLTDPTH= $54
#BLTDPTL= $56
```

Each pair of registers contain an 18 bit pointer to the start address of the 4 blitter channels in chip mem.

```
#BLTAMOD=$64
#BLTBMOD=$62
#BLTCMOD=$60
#BLTDMOD=$66
```

The 4 modulo values are added to the blitter pointers at the end of each line.

```
#BLTADAT=$74
#BLTBDAT=$72
#BLTCDAT= $70
```

If a blitter channel is disabled the BLTxDAT register can be loaded with a constant value which will remain unchanged during the blit operation.

```
#BLTAFWM=$44   ; Blitter first word mask for source A
#BLTALWM=$46   ; Blitter last word mask for source A
```

During a Blitter operation these two registers are used to mask the contents of BLTADAT for the first and last word of every line.

#BLTCON0=$40
#BLTCON1=$42

The following bits in BLTCON0 & BLTCON1 are as follows.

BIT#	BLTCON0	BLTCON1
15	ASH3	BSH3
14	ASH2	BSH2
13	ASH1	BSH1
12	ASH0	BSH0
11	USEA	x
10	USEB	x
09	USEC	x
08	USED	x
07	LF7	x
06	LF6	x
05	LF5	x
04	LF4	EFE
03	LF3	IFE
02	LF2	FCI
01	LF1	DESC
00	LF0	0 (1=line mode)

ASH is the amount that source A is shifted (barrel rolled)
USEx enables each of the 4 blitter channels
LF holds the logic function as discussed previously in this section
BSH is the amount that source B is shifted (barrel rolled)
EFE is the Exclusive Fill Enable flag
IFE is the Inclusive Fill Enable flag
FCI is the Fill Carry Input
DESC is the descending flag (blitter uses decreasing addressing)

#BLTSIZE=$58

By writing the height and width of the blit operation to BLTSIZE the the blitter will start the operation. Maximum size is 1024 high and 64 words (1024 bits) wide. The following defines bits in BLITZSIZE

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	h9	h8	h7	h6	h5	h4	h3	h2	h1	h0	w5	w4	w3	w2	w1	w0

#BLTSIZV= $5C *;(ECS ONLY)*
#BLTSIZH =$5C *;(ECS ONLY)*

With the new ECS writing to BLTSIZV first and then BLTSZH the blitter can operate on blocks as large as 32K x 32K pixels in size.

The Blitter is also able to perform linedrawing and filled polygon functions. Details about using the blitter for these functions can be found on the examples disk included with Blitz 2.

Audio Control

The Amiga has 4 channels of 8 bit audio, each with their own memory access, period and volume control. The following are a list of the applicable hardware registers.

```
#AUD0LCH=$A0  ;pairs of 24 bit memory pointers to audio data in chip mem
#AUD0LCL=$A2
#AUD1LCH=$B0
#AUD1LCL=$B2
#AUD2LCH=$C0
#AUD2LCL=$C2
#AUD3LCH=$D0
#AUD3LCL=$D2

#AUD0LEN=$A4  ;volume registers (0-63)
#AUD1LEN=$B4
#AUD2LEN=$C4
#AUD3LEN=$D4

#AUD0PER=$A6  ;period
#AUD1PER=$B6
#AUD2PER=$C6
#AUD3PER=$D6

#AUD0VOL=$A8
#AUD1VOL=$B8
#AUD2VOL=$C8
#AUD3VOL=$D8

#AUD0DAT=$AA
#AUD1DAT=$BA
#AUD2DAT=$CA
#AUD3DAT=$DA
```

Sprite Control

The Amiga hardware is capable of displaying eight 4 colour sprites or four 16 colour sprites. Standard control of sprites is done by using the copper to setup the 8 sprite pointers at the beginning of each frame.

```
#SPR0PTH=$120  ;pairs of 24 bit memory pointers to sprite data in chip mem
#SPR0PTL=$122
#SPR1PTH=$124
#SPR1PTL=$126
#SPR2PTH=$128
#SPR2PTL=$12A
#SPR3PTH=$12C
#SPR3PTL=$12E
#SPR4PTH=$130
#SPR4PTL=$132
#SPR5PTH=$134
#SPR5PTL=$136
#SPR6PTH=$138
#SPR6PTL=$13A
#SPR7PTH=$13C
#SPR7PTL=$13E
```

The pointers should point to data that is begins with two words containing the SPRPOS & SPRCTL values for that sprite, followed by its image data and with two null words that terminate the data.

#SPR0POS = $140	#SPR0CTL = $142	#SPR0DATA = $144	#SPR0DATB = $146
#SPR1POS = $148	#SPR1CTL = $14A	#SPR1DATA = $14C	#SPR1DATB = $14E
#SPR2POS = $150	#SPR2CTL = $152	#SPR2DATA = $154	#SPR2DATB = $156
#SPR3POS = $158	#SPR3CTL = $15A	#SPR3DATA = $15C	#SPR3DATB = $15E
#SPR4POS = $160	#SPR4CTL = $162	#SPR4DATA = $164	#SPR4DATB = $166
#SPR5POS = $168	#SPR5CTL = $16A	#SPR5DATA = $16C	#SPR5DATB = $16E
#SPR6POS = $170	#SPR6CTL = $172	#SPR6DATA = $174	#SPR6DATB = $176
#SPR7POS = $178	#SPR7CTL = $17A	#SPR7DATA = $17C	#SPR7DATB = $17E

Using standard sprite DMA the above registers are all loaded from the sprite data pointed to in chip mem by the sprite pointers. These registers are only of interest to people wanting to 'multiplex' sprites by using the copper to load these registers rather than sprite DMA.

The following is bit definitions of both SPRPOS and SPRCTL.

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
POS	SV7	SV6	SV5	SV4	SV3	SV2	SV1	SV0	SH8	SH7	SH6	SH5	SH4	SH3	SH2	SH1
CTL	EV7	EV6	EV5	EV4	EV3	EV2	EV1	EV0	ATT	X	X	X	X	SV8	EV8	SH0

SV is the vertical start position of the sprite
SH is the horizontal position of the sprite (calulated in lores pixels only)
EV is the end vertical position
ATT is the sprite attached bit (connects odd sprites to their predecessors)

Interupt Control

```
#INTENA=$9A    ;interupt enable write address
#INTENAR=$1C  ;interupt enable read address

#INTREQ=$9C    ;interupt request write address
#INTREQR=$1e  ;interupt request read address
```

INTENA is used to enable or disable interupts. If the value written to INTENA has bit 15 set any other of the bits enable their corresponding interupts. If bit 15 is clear any of the other bits set will disable their corresponding interupts.

INTENAR will return which interupts are currently enabled.

INTREQ is used to initiate or clear an interupt. It is mostly used to clear the interupt by the interupt handler. Again Bit# 15 states whether the corrsponding interupts will be requested or cleared.

INTREQR returns which interupts are currently requested.

The following bit definitions relate to the 4 interupt control registers.

BIT#	NAME	LEVEL	DESCRIPTION
15	SET/CLR		determines if bits written with 1 are set or cleared
14	INTEN		master interupt enable
13	EXTER	6	external interupt
12	DSKSYN	5	disk sync register (same as DSKSYNC)
11	RBF	5	serial port Receive Buffer Full
10	AUD3	4	audio channel 3 finished
09	AUD2	4	audio channel 2 finished
08	AUD1	4	audio channel 1 finished
07	AUD0	4	audio channel 0 finished
06	BLIT	3	blitter finished
05	VERTB	3	start of vertical blank interupt
04	COPER	3	copper
03	PORTS	2	I/O ports and timers
02	SOFT	1	reserved for software initiated interupts
01	DSKBLK	1	disk block finished
00	TBE	1	serial port Transmit Buffer Empty

The following locations hold the address of the 68000 interupt handler code in memory for each level of interupt.

LEVEL	68000 Address
6	$78
5	$74
4	$70
3	$6c
2	$68
1	$64

DMA Control

DMA stands for direct memory access. Chip mem can be accessed by the display, blitter, copper, audio, sprites and diskdrive without using the 68000 processor. DMACON enables the user to lock out any of these from havihg direct memory access (DMA) to chipmem.

As with INTENA bit 15 of DMACON signals whether the write opertaion should clear or set the relevant bits of the DMA control.

DMACONR will not only return which channels have DMA access but has flags BBUSY which return true if the blitter is in operation and BZERO which return if the Blitter has generated any 1's from it logic function (useful for collision detection etc.)

```
#DMACON=$96    ;DMA control write (clear or set)
#DMACONR=$02   ;DMA control read (and blitter status) read
```

The following are the bits assigned to the two DMACON registers:

BIT#	NAME	DESCRIPTION
15	SET/CLR	determines if bits written with 1 are set or cleared
14	BBUSY	blitter busy flag
13	BZERO	blitter logic zero
12	X	
11	X	
10	BLTPRI	"blitter nasty" signals blitter has DMA priority over CPU
09	DMAEN	enable all DMA below
08	BPLEN	BitPlane DMA enable
07	COPEN	Copper DMA enable
06	BLTEN	Blitter DMA enable
05	SPREN	Sprite DMA enable
04	DSKEN	Disk DMA enable
03	AUD3EN	Audio channel 3 DMA enable
02	AUD2EN	Audio channel 2 DMA enable
01	AUD1EN	Audio channel 1 DMA enable
00	AUD0EN	Audio channel 0 DMA enable

Amiga CIAs

The Amiga has two 8520 Complex Interface Adapter (CIA) which handle most of the Amiga I/O activities. Note that each register should be accessed as a byte and NOT a word. The following is an address map of both Amiga CIAs.

CIA-A	Address	Register	b7	b6	b5	b4	b3	b2	b1	b0
	$BFE001	pra	FIR1	FIR0	RDY	TK0	WPR0	CHNG	LED	OVL
	$BFE101	prb	Parallel Port							
	$BFE201	ddra	Direction for Port A (1=output)							
	$BFE301	ddrb	Direction for Port B (1=output)							
	$BFE401	talo	Timer A High Byte							
	$BFE501	tahi	Timer A High Byte							
	$BFE601	tblo	Timer B Low Byte							
	$BFE701	tbhi	Timer B High Byte							
	$BFE801	todlo	50/60 Hz Event Counter bits 7-0							
	$BFE901	todmid	50/60 Hz Event Counter bits 15-8							
	$BFEA01	todhi	50/60 Hz Event Counter bits 23-16							
	$BFEB01		not used							
	$BFEC01	sdr	Serial Data Register (connected to keyboard)							
	$BFED01	icr	Interrupt Control Register							
	$BFEE01	cra	Control Register A							
	$BFEF01	crb	Control Register B							

CIA-B	Address	Register	b7	b6	b5	b4	b3	b2	b1	b0
	$BFD000	pra	DTR	RTS	CD	CTS	DSR	SEL	POUT	BUSY
	$BFD100	prb	MTR	SEL3	SEL2	SEL1	SEL0	SIDE	DIR	STEP
	$BFD200	ddra	Direction for Port A (1=output)							
	$BFD300	ddrb	Direction for Port B (1=output)							
	$BFD400	talo	Timer A High Byte							
	$BFD500	tahi	Timer A High Byte							
	$BFD600	tblo	Timer B Low Byte							
	$BFD700	tbhi	Timer B High Byte							
	$BFD800	todlo	Horizontal Sync Event Counter bits 7-0							
	$BFD900	todmid	Horizontal Sync Event Counter bits 15-8							
	$BFDA00	todhi	Horizontal Sync Event Counter bits 23-16							
	$BFDB00		not used							
	$BFDC00	sdr	Serial Data Register (connected to keyboard)							
	$BFDD00	icr	Interrupt Control Register							
	$BFDE00	cra	Control Register A							
	$BFDF00	crb	Control Register B							

APPENDIX 4: 68000 ASSEMBLY LANGUAGE

Although Blitz 2 is a BASIC compiler, it also has an 'inline assembler' and can be used as a fully fledged assembler. Assembly language is the language of the microprocessor, in the case of the Amiga, the 68000 microprocessor.

The following is a brief description of the Motorola 68000 microprocessor and its instruction set, for more information we recommend the data books published by Motorola themselves as the best source of reference material.

Registers

The 68000 has 16 internal registers, these may be thought of as high speed variables each capable of storing a long word (32 bits). The 8 data registers are used mainly for calculations while the 8 address registers are mostly used for pointing to locations in memory.

The registers are named D0-D7 and A0-A7. The 68000 also has several specialised registers, the program counter (PC) and the status register (SR). The program counter points to the current instruction that the microprocessor is executing, while the status register is a bunch of flags with various meanings.

Addressing

The main job of the microprocessor is to read information from memory, perform a calculation and then write the result back to memory.

For the processor to access memory it has to generate a memory address for the location it wishes to access (read or write to). The following are the different ways the 68000 can generate addresses.

Register Direct: MOVE d1,d0
The actual value in the register d1 is copied into d0

Address Register Indirect: MOVE (a0),d0
a0 is a pointer to somewhere in memory. The value at at this location is copied into the register d0.

Address Register Indirect with Postincrement: MOVE (a0)+,d0
The value at the location pointed to by a0 is copied into the register d0, then a0 is incremented so it points to the next memory location.

Address Register Indirect with Predecrement: MOVE -(a0),d0
a0 is first decremented to point to the memory location before the one it currently points to then the value at the new memory location is copied into d0.

Address Register Indirect with Displacement: MOVE 16(a0),d0
The memory location located 16 bytes after that which is pointed to by address register a0 is copied to d0.

Address Register Indirect with Index: MOVE 16(a0,d1),d0
The memory location is calculated by adding the contents of a0 with d1 plus 16.

Absolute Address: MOVE $dff096,d0
The memory location $dff096 is used.

Program Counter with Displacement: MOVE label(pc),d0
This is the same as absolute addressing but because the memory address is an offset from the program counter (no bigger than 32000 bytes) it is MUCH quicker.

Program Counter with Index: MOVE label(pc,d1),d0
The address is calulated as the location of label plus the contents of data register d1.

Immediate Data: MOVE #20,d0
The value 20 is moved to the data register.

Program Flow

As mentioned previously the microprocessor has a special register known as the program counter that points to the next instruction to be executed. By changing the value in the program counter a 'goto' can be performed. The JMP instruction load the program counter with a new value, it supports most of the addressing modes.

A branch is a program counter relative form of the JMP instruction. Branches can also be performed on certain conditions such as BCC which will only cause the program flow to change if the Carry flag in the status register is currently set.

A 'gosub' can be prformed using the JSR and BSR commands. The current value of the program counter is remembered on the stack before the jump or branch is performed. The RTS command is used to 'return' to the original program location.

The Stack

The Amiga sets aside a certain amount of memory for each task known as a stack. The address register A7 is used to point to the stack and should never be used as a general purpose address register.

The 68000 uses predecrement addressing to push data onto the stack and postincrement addressing to pull information off the stack.

JSR is the same as MOVE.l pc,-(a7) and then JMP

RTS is the same as MOVE.l (a7)+,pc

The stack can be used to temporarily store internal registers. To save and restore all the 68000 registers the following code is often used

```
ASubroutine:
        MOVEM.l d0-d7/a0-a6,-(a7)   ;push all register on stack
        ;main subroutine code here which can stuff up registers without worrying
        MOVEM.l (a7)+,d0-d7/a0-a6  ;pull registers off stack
        RTS                         ;returnfrom subroutine
```

Condition Flags

The status register is a special 68000 register that holds, besides other things all the condition codes. The following are a list of the condition flags:

Code	Name	Meaning
N	negative	reflects the most significant bit of the result of the last operation.
Z	zero	is set if the result is zero, cleared otherwise.
C	carry	is set when an add, subtract or compare operation generate a carry
X	extend	is a mirror of the carry flag, however its not affected by data movement.
V	overflow	is set when an aritmeitic operation causes an overflow, a situation where the operand is not large enough to represent the result.

Conditional Tests

Branches and Sets can be performed conditionally. The following is a list of the possible conditions that can be tested before a branch or set is performed.

cc	condition	coding	test
T	true	0000	1
F	false	0001	0
HI	high	0010	not C & not Z
LS	lowsam	0011	C I Z
CC	carry clr	0100	not C
CS	carry set	0101	C
NE	ot equal	0110	not Z
EQ	equal	0111	Z
VC	overflow clr	1000	not V
VS	overflow set	1001	V
PL	plus	1010	not N
MI	minus	1011	N
GE	greater equal	1100	N&V I notN&notV
LT	less than	1101	N&notV I notN&V
GT	greater than	1110	N&V&notZ I notN&notV&notC
LE	less or equal	1111	Z I N&notV I notN&V

Operand Sizes

The 68000 can perform operations on bytes, words and long words. By adding a suffix .b .w or .l to the opcode, the assembler knows which data size you wish to use, if no suffix is present the word size is default. There is no speed increase using bytes instead of words as the 68000 is a 16 bit microprocessor and so no overhead is needed for 16 bit operations. However 32 bit long words do cause overhead with extra read and write cycles needed to perform operations on a bus that can only handle 16 bits at a time.

The 68000 Instruction Set

The following is a brief description of the 68000 instruction set.

Included with each are the addrssing mode combinations available with each opcode. Their syntax are as follows:

Dn	data register
An	address register
Dy,Dx	data registers source & destination
Rx,Ry	register source & destination (data & address registers)
<ea>	effective address - a subset of addressing modes
#<data>	numeric constant

Special notes:

The address register operands ADDA, CMPA, MOVEA and SUBA are only word and long word data sizes. The last 'A' of the operand name is optional as it is with the immediate operands ADDI, CMPI, MOVEI, SUBI, ORI, EORI and ANDI.

The ADDQ and SUBQ are quick forms of their immediate cousins. The immediate data range is 1 to 8. The MOVEQ instruction has a data range of -128 to 127, the data is sign extended to 32 bits, and long is the only data size available.

The <ea> denotes an effective address, not all addressing modes are available with each effective address form of the instruction, as a rule program counter relative addressing is only available for the sourse operand and not the destination.

The Blitz2 compiler will signal any illegal forms of the instruction during the compile stage.

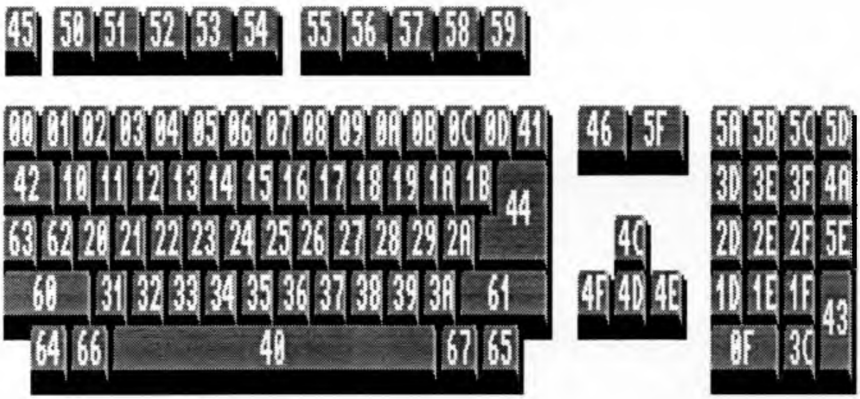
ABCD	*Add with extend using BCD*
	ABCD Dy,Dx
	ABCD -(Ay),-(Ax)
ADD	*Add binary*
	ADD <ea>,Dn
	ADD Dn,<ea>
	ADDA <ea>,An
	ADDI #<data>,<ea>
	ADDQ #<data>,<ea>
ADDX	*Add with Extend*
	ADDX Dy,Dx
	ADDX -(Ay),-(Ax)
AND	*AND logical*
	AND <ea>,Dn
	AND Dn,<ea>
	ANDI #<data>,<ea>
ASL	*Arithmetic Shift Left*
	ASL Dx,Dy
	ASL #<data>,Dy
	ASL <ea>
ASR	*Arithmetic Shift Right*
	ASR Dx,Dy
	ASR #<data>,Dy
	ASR <ea>
Bcc	*Branch Conditionally*
	Bcd <label>
BCHG	*Test a Bit & Change*
	BCHG Dn,<ea>
	BCHG #<data>,<ea>
BCLR	*Test a Bit & Clear*
	BCLR Dn,<ea>
	BCLR #<data>,<ea>
BRA	Branch Always
	BRA <label>
BSET	*Test a Bit & Set*

	BSET Dn,<ea>
	BSET #<data>,<ea>
BTST	*Test a Bit*
	BTST Dn,<ea>
	BTST #<data>,<ea>
CHK	*Check Register Against Bounds*
	CHK <ea>,Dn
CLR	Clear an Operand
	CLR <ea>
CMP	*Compare*
	CMP <ea>,Dn
	CMPA <ea>,An
	CMPI #<data>,<ea>
CMPM	*Compare Memory*
	CMPM (Ay)+,(Ax)+
DBcc	*Test Condition, Decrement, and Branch*
	DBcc Dn,<label>
DIVS	*Signed Divide*
	DIVS <ea>,Dn Data
DIVU	*Unsigned Divide*
	DIVU <ea>,Dn
EOR	*Exclusive OR Logical*
	EOR Dn,<ea>
	EORI #<data>,<ea>
EXG	*Exchange Registers*
	EXG Rx,Ry
EXT	*Sign Extend*
	EXT Dn Data
ILLEGAL	*Illegal Instruction*
	ILLEGAL
JMP	*Jump*
	JMP <ea>
JSR	*Jump to Subroutine*
	JSR <ea>
LEA	*Load Effective Address*
	LEA <ea>,An
LINK	*Link and Allocate*
	LINK An,#<displacement>
LSL	*Logical Shift Left*
	LSL Dx,Dy
	LSL #<data>,Dy
	LSL <ea>
LSR	*Logical Shift Right*
	LSR Dx,Dy
	LSR #<data>,Dy
	LSR <ea>
MOVE	*Move Data from Source to Destination*
	MOVE <ea>,<ea>
	MOVEA <ea>,An
	MOVEQ #<data>,Dn
MOVEM	*Move Multiple Registers*
	MOVEM <register list>,<ea>
	MOVEM <ea>,<register list>
MOVEP	*Move Peripheral*
	MOVEP Dx,d(Ay)
	MOVEP d(Ay),Dx
MULS	*Signed Multiple*
	MULS <ea>,Dn
MULU	*Unsigned Multiple*
	MULU <ea>,Dn
NBCD	*Negate Decimal with Extend*
	NBCD <ea>
NEG	*Negate*
	NEG <ea>
NEGX	*Negate with Extend*

	NEGX <ea>
NOP	*No Operation*
	NOP
NOT	*Logical Complement*
	NOT <ea>
OR	*Inclusice OR Logical*
	OR <ea>,Dn
	OR Dn,<ea>
	ORI #<data>,<ea>
PEA	*Push Effective Address*
	PEA <ea>
RESET	*Reset External Device*
	RESET
ROL	*Rotate Left (without Extend)*
	ROL Dx,Dy
	ROL #<data>,Dn
	ROL <ea>
ROR	*Rotate Right (without Extend)*
	ROR Dx,Dy
	ROR #<data>,Dn
	ROR <ea>
ROXL	*Rotate Left with Extend*
	ROXL Dx,Dy
	ROXL #<data>,Dn
	ROXL <ea>
ROXR	*Rotate Right with Extend*
	ROXR Dx,Dy
	ROXR #<data>,Dn
	ROXR <ea>
RTE	*Return from Exception*
	RTE Data
RTR	*Return and Restore Condition Codes*
	RTR
RTS	*Return from Subroutine*
	RTS
SBCD	*Subtract Decimal with Extend*
	SBCD Dy,Dx
	SBCD -(Ay),-(Ax)
Scc	*Set according to Condition*
	Scc <ea>
STOP	*Load Status Register and Stop*
	STOP #xxx
SUB	*Subtract Binary*
	SUB <ea>,Dn
	SUB Dn,<ea>
	SUBA <ea>,An
	SUBI #<data>,<ea>
	SUBQ #<data>,<ea>
SUBX	*Subtract with Extend*
	SUBX Dy,Dx
	SUBX -(Ay),-(Ax)
SWAP	*Swap Register Halves*
	SWAP Dn
TAS	*Test & Set an Operand*
	TAS <ea>
TRAP	Trap
	TRAP #<vector>
TRAPV	*Trap an Overflow*
	TRAPV
TST	*Test an Operand*
	TST <ea>
UNLK	*Unlink*
	UNLK An Data

APPENDIX 5: RAW KEY CODES

The following keycodes are in hex, and are for use with the blitzmode command RawKeyStatus.



COMMAND INDEX

Blitz Basic Addendum

Page #	Line #	Mistake Type	Change from	Change to
5	7	Font/Insert	INSERTFROMDISKL	INSERTFROMDISK Loads
6	3,5,14	Spelling	3 curosr 5 curosr 14 lettters	3 cursor 5 cursor 14 letters
10	4	Spelling	avaiiable	available
11	16	Spelling	calulation	calculation
17	25	Font	or	**OR**
19	**27**	Delete	**MouseWait Next**	**MouseWait** *Delete all text after the first MouseWait*
25	30	Spelling	addess.s	address.s
28	**13**	Insert	flags.w x y	flags.w:x:y
29	24,25, **29**	Font/Insert	24 If...Then...Else 25 If...Then...Else 29 comes to the	24 **If...Then...Else** 25 **If...Then...Else** **29** comes to the end of the list.
30	3,8,21	Font/Delete	3 **Killltem** 8 **Killltem** 21 given the the new	3 **Killltem** 8 **Killltem** 21 given the new
31	16	Font	Statements	**<u>Statements</u>**
34	**13**	Delete	The following code is an example of an assembler procedure in Blitz:	For an example of an assembler procedure in Blitz, turn to page 63.
38	23	Spelling	it's	its'
39	5	Spelling	Igonore	Ignore
42	18	Spelling	Pallettes	Palettes
45	5	Font/Insert	Free Bitmap	**Free Bitmap** *n*
55	36	Spelling	adress-4	address-4
60	6	Spelling	is	if
63	**6,7**	Delete	Because address register a4 is used as the local variable base, the UNLK a4 command must be placed at the top of a procedure,	*Delete this sentence as UNLK a4 is no longer required. AsmExit is used at the bottom of an assembler procedure for this purpose.*

Page #	Line #	Mistake Type	Change from	Change to
68	16	Font	Keeping your code readable.	**Keeping your code readable.**
71	17,**24**, **26**	Insert	17 codeas **24** $f00 **26** $f00	17 code as **24** #$f00 **26** #$f00
82	15	Spelling	charcter	character
83	13	Spelling	simple take	simply takes
85	2	Spelling	demonstrate	demonstrates
89	31	Spelling/Insert	tha #agacolors	that the #agacolors
91	16	Spelling	3 colours sprited	3 coloured sprites
92	15	Spelling	poistive	positive
94	6	Spelling	againg	again
100	25,38	Spelling/Insert	25 attached 38 will reported	25 attach 38 will be reported
103	8	Font	**Killltem**	**KillItem**
104	**13**	Delete	Only primitive type, `non-list` arrays may be sorted; it is not possible to sort newtype arrays, or `list` arrays.	*Delete this line. List arrays can be sorted using the SortList command.*
106	37	Spelling	character	characters
107	4	Delete	program user.	
108	9	Spelling	beng	being
114	13,**33**	Insert/Delete	13 and less or **33 Tan Tan*(float)***	13 and less than or **33 Tan *(float)***
115	26	Insert	Length leftmost	the leftmost Length
118	1	Insert	at beginning	at the beginning
130	10	Spelling	mystuff/myprogs	mystuff/myprograms
131	11	Spelling	lice	slice
135	8	Insert	commanddoes	command does
140	17	Spelling	will	when
141	8,14,30	Spelling	8 using 14 using 30 using	8 used 14 used 30 used
142	31	Insert/Spelling	of eigth	of an eighth
145	21	Spelling	commands	command
146	27	Spelling	draw	draws
153	4	Spelling	animations	animation

Page #	Line #	Mistake Type	Change from	Change to
156	20	Insert	to BBlited	to be BBlited
157	42	Spelling/Insert	1bcan shapeyou	can shape you
159	38	Spelling	display	displayed
161	12	Delete	Bitplanes refers to the number of bitplanes (depth) of the bitmap collisions are to be tested for in.	Bitplanes refers to the number of bitplanes (depth) that bitmap collision are to be tested in.
162	5	Insert	channel the sprite	channel of the sprite
164	**21**	Delete	Use Palette	ShowPalette *Use Palette no longer copies a palettes colours onto the display. The commands ShowPalette and DisplayPalette are for this purpose.*
165	**18**	Delete	Use Palette	ShowPalette See explanation above.
166	26,31	Insert	26 thewhole 31 there any steps	26 the whole 31 there are any steps
172	10,**22**	Insert/Delete	10 memorylocation **22** Translate	10 memory location **22** PhoneticSpeak command
173	**25**,31	Insert/Spelling	**25** Omission from description 31 own	**25** The Dpen parameter is the detail pen colour used for the screen title. The Bpen parameter is the block pen colour used for the screens' borders. 31 on

Page #	Line #	Mistake Type	Change from	Change to
174	7	Delete	Use Palette	ShowPalette *See explanation for error on page 164 (above).*
175	7	Delete	(see autodocs/	
176	34	Delete	you have may want	you may want
180	10,20	Spelling	10 return 20 Supplyng	10 returns 20 Supplying
182	13,20, 21	Spelling/Insert	13 clour 13 or colour 20 Cls 21 If colour 21 specfied	13 colour 13 or a colour 20 WCls 21 If a colour 21 specified
183	5,7,11, 29,32	Spelling/Insert	5 editting 7 editting 11 editting 29 gaphics 32 leavesthe	5 editing 7 editing 11 editing 29 graphics 32 leaves the
184	7	Spelling	active	activate
186	11	Spelling	***(winodw#)***	***(window#)***
187	22	Delete/Spelling Insert	22 descibed 22 descibed in	22 described 22 described in the
188	25,27	Delete/Spelling	25 as so should 27 witdh	25 as should 27 width
189	6	Font	hello	HELLO
190	32	Spelling	gadgets	gadget.
191	38	Delete	may be be	may be
194	11,12	Spelling/Delete	11 Togggle 12 if the no On	11 Toggle 12 if the On
196	20,28	Spelling	20 grpahics 28 pixles	20 graphics 28 pixels
199	8,17,26	Spelling/Insert	8 enaables 17 readonly 26 of display	8 enables 17 read only 26 of the display
200	3	Spelling	AttchGTList	AttachGTList
201	4,29	Spelling	4 reattache 29 of and gadtools	4 reattach 29 of the gadtools

Page #	Line #	Mistake Type	Change from	Change to
202	**21**	Insert	The following code (....etc.)	*Insert the code shown at the end of this error report.*
204	21	Insert	one ormore of	one or more of
205	29	Insert	theLONG	the LONG
206	18	Font	*the*	the
208	22	Spelling	tht	that
213	23	Spelling	BRex	BRexx
214	6	Delete/Spelling	and unit 2 or refer to	and unit 2 refers to
215	25	Spelling/Insert	25 send 25 out the given	25 sends 25 out to the given
217	22,23	Spelling	22 Blitz][23 Parmaters	22 Blitz2 23 Parameters
218	4	Insert	may only defined	may only be defined
219	24	Spelling	Blitz][	Blitz2
222	16	Insert	wellas	well as
223	10,12, 19,21	Insert	10 SetInt..SetInt 12 SetInt..SetInt 19 SetInt..SetInt 21 SetInt..SetInt	10 SetInt..End SetInt 12 SetInt..End SetInt 19 SetInt..End SetInt 21 SetInt..End SetInt
224	5	Spelling	Blitz][	Blitz2
240	29	Spelling	there	their
242	30,32, 36	Delete/Spelling	30 the the blitter 32 BLITZSIZE 36 #BLTSIZH=$5C	30 the blitter 32 BLTZSIZE 36 #BLTSIZH=$5E
243	48	Delete	data that is begins	data that begins
244	25	Font	$1e	$1E
245	37	Spelling	it	its'
247	25	Delete	The value at at this	The value at this
248	18,23	Spelling	18 load 23 prformed	18 loads 23 performed
249	9,21	Spelling	9 aritmetic 21 ot	9 arithmetic 21 not
250	3,20	Spelling	3 addrsing 20 sourse	3 addressing 20 source
252	6	Spelling	Inclusice	Inclusive
Back Page	23	Spelling	appeard	appeared

Example code missing from page 202 (ASLFontRequest):

```
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\pen1
	NPrint *f\pen2
	NPrint *f\drawmode
Else
	NPrint "cancelled"
EndIf

MouseWait
```

Copyright

Disclaimer

Acid Software accept no responsibility for the performance of Blitz2 or software developed with Blitz2. However, we will endeavour to solve any problems registered users have with the product and hope to support the product to the best of our ability.

Credits

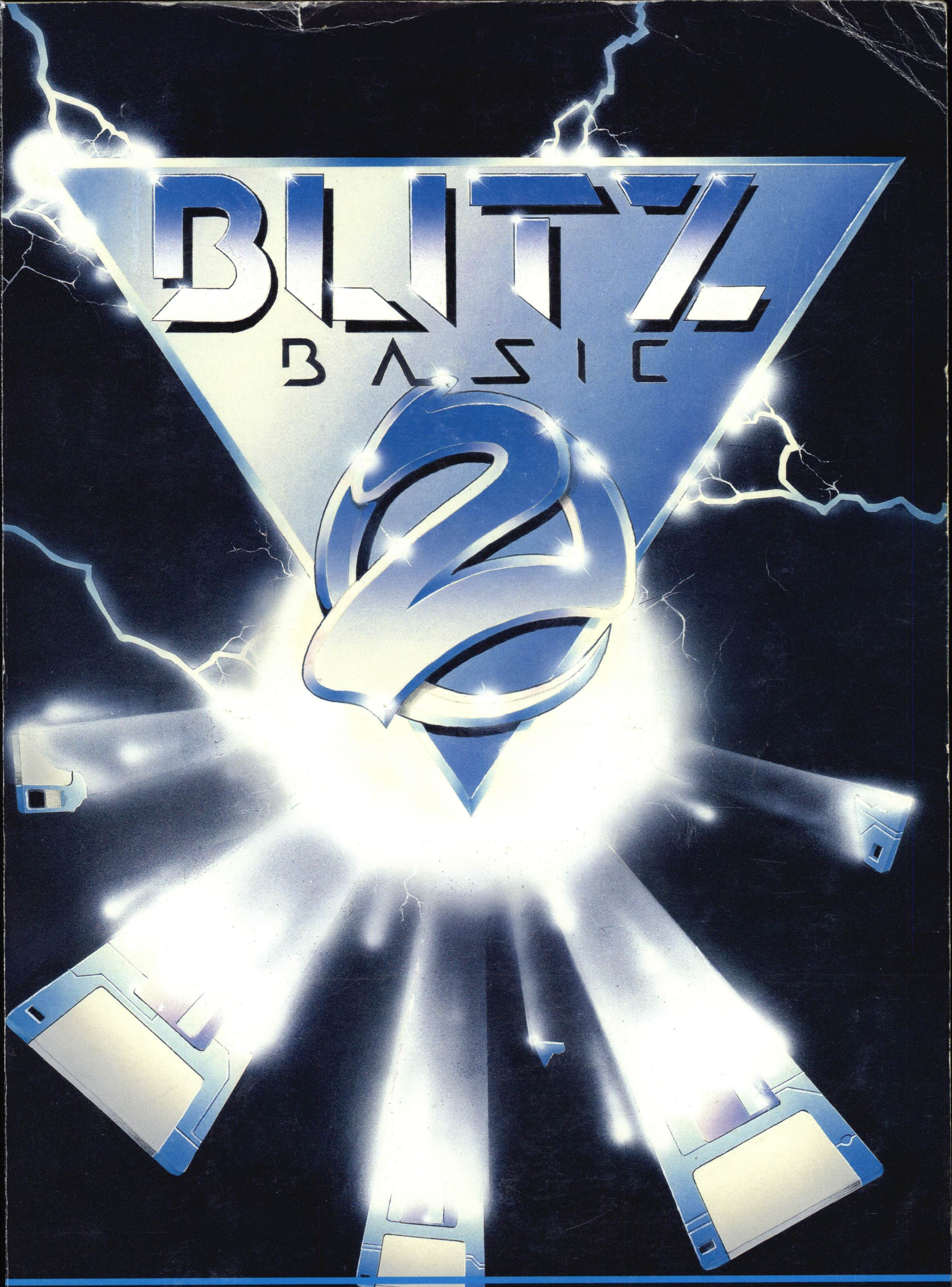
Developed by Mark Sibly using HiSoft's Devpac2.
Printed in the United Kingdom by Guildhall Leisure Services
Manual produced with Soft Logik's PageStream2 & PageLiner.

Technical Support

Users are not entitled to any support unless registration cards are returned. Support is available via email from acid@iconz.co.nz. Acid Software can also be contacted via fax in New Zealand on +649 358 1658. Alternatively mail any questions to the address that appeard on the registration card enclosed in this package.

Upgrades

Blitz BASIC for the Amiga is continually being improved. Updates are provided free by subscribing to the Blitz User Magazine (BUM), details are included on the registration card. Significant additions to the language are planned for 1995 including Atari Jaguar support, 3D graphics support for both Amiga and Jaguar, as well as a powerful new editor.



ACID SOFTWARE

REFERENCE MANUAL

CONTENTS

Chapters

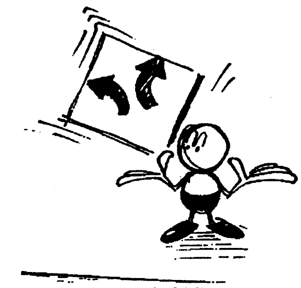
Appendicies

Blitz BASIC 2 was developed by Mark Sibly

COPYRIGHT

Printed in Auckland, New Zealand by UNIPRINT

1. Program Flow



Program flow refers to the order in which a program's instructions are executed. When a program is run, it's commands are executed in a top-down manner. This means instructions are executed one after another, from the top of your program to the bottom. This section deals with commands which interupt this normal process, and cause commands to be executed from a different point in the program.

Amiga interupt control commands are also covered at the end of this section.

Statement: **Goto**

Syntax: Goto *Program Label*

Modes: Amiga/Blitz

Description:

Goto causes program flow to be transferred to the specified *program label*. This allows sections of a program to be 'skipped' or 'repeated'.

Example:

```
;
; goto program example
;

Goto there
NPrint "What happened to me?"

there:
NPrint "Program flow has reached 'there'"
MouseWait
```

See Also:

Gosub

Statement: **Gosub**

Syntax: Gosub *Program Label*

Modes: Amiga/Blitz

Description:

Gosub 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 is usually terminated by a **Return** command. The **Return** command has the effect of returning program flow to the location remembered by the previous **Gosub** command.

This allows a section of a program to be used by many other parts of the same program.

Example:

```
;
; gosub program example
;

Gosub two
NPrint "Three!"
MouseWait
End

two:
Gosub one
NPrint "Two!"
Return

one:
NPrint "One!"
Return
```

See Also:

Return

Statement: **Return**

Syntax: Return

Modes: Amiga/Blitz

Description:

Return is 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.

See Also:

Gosub

Statement: **On...Goto| Gosub**

Syntax: On *Expression* ***Goto| Gosub*** *Program Label[,Program Label...]*

Modes: Amiga/Blitz

Description:

On 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 *Program Label* will be branched to. A result of 2 will cause the second *Program Label* to be branched to and so on.

If the result of *Expression* is less than one, or not enough *Program Labels* are supplied, program flow will continue from the command following the **On**.

Example:

```
;
; on...gosub prgram example
;

For k=1 To 3
 On k Gosub one,two,three
Next

MouseWait
End

one:
NPrint "One!"
Return

two:
NPrint "Two!"
Return

three:NPrint "Three!"
Return

NPrint "Click mouse button to return to the editor..."
MouseWait
```

Statement: **MouseWait**

Syntax: MouseWait

Modes: Amiga/Blitz

Description:

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

This is often useful in Blitz 2 to prevent a program from terminating too quickly and leaving you back in the editor.

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

Example:

```
;
; mousewait program example
;

a=10
NPrint "Click mouse button, then type 'NPrint a'"
MouseWait
Stop
```

Statement: End

Syntax: End

Modes: Amiga/Blitz

Description:

End will halt program flow completely. In the case of programs run from the Blitz 2 editor, you will be returned to the editor. In the case of executable files, you will be returned to the Workbench or CLI.

End is often also useful to prevent program flow from running into a subroutine.

Example:

```
;
; end program example
;

Gosub there
MouseWait
End
there:
NPrint "Hello!"
Return
```

See Also:

Stop

Statement: Stop

Syntax: Stop

Modes: Amiga/Blitz

Description:

The **Stop** command will cause program flow to stop, and user control to be transferred to Blitz 2 direct mode.

The **Stop** command is really only useful in debugging situations, as it allows the programmer a chance to have a look at program variables via Blitz 2's direct mode.

Example:

```
;
; stop program example
;

a=10
NPrint "Click mouse button, then type 'NPrint a'"
MouseWait
Stop
```

See Also:

End, Cont

Statement: **Cont**

Syntax: **Cont** *[N]*

Modes: Amiga/Blitz

Description:

The **Cont** command is only available in Blitz 2 direct mode. **Cont** will cause program flow to continue from the instruction following the instruction which caused a jump to direct mode. This instruction may have been either a **Stop** or a program error of some kind.

The optional *N* parameter can be used to tell Blitz 2 programs to ignore a number of **Stop** commands after a **Cont**. This is useful in debugging as it allows you to insert a **Stop** inside a program loop, but not have to **Cont** every pass of the loop.

See Also:

Stop

Statement: **If**

Syntax: If *Expression [Then...]*

Modes: Amiga/Blitz

Description:

If allows you to execute a section of program depending on the value of program variables. *Expression* usually includes some form of comparison operator.

If an **If** is followed by a **Then**, and the expression proves to be true, then the instructions following the **Then** will be executed. If the expression proves to be false, then the instructions following the **Then** are ignored, and program flow continues from the line following the **If**.

If an **If** is NOT followed by a **Then**, and the expression proves to be true, then program flow will continue from the instruction following the **If**. If the expression proves to be false, then program flow will continue from the instruction following the next matching **EndIf** or **Else** command. Blocks of program instructions inside an **If** and an **EndIf** are known as 'If blocks'.

Example:

```
;
; if...then program example
;

For k=1 To 10
 If k=5 Then NPrint "k is 5!"
 If k<5
  NPrint "k is less than 5!"
 Else
  NPrint "k is not less than 5!"
 EndIf
Next
MouseWait
```

See Also:

Else, EndIf

Statement: EndIf

Syntax: EndIf

Modes: Amiga/Blitz

Description:

EndIf is used to terminate an 'If block'. An If block is begun by use of the **If** statement. Please refer to **If** for more information on If blocks.

See Also:

If, Else

Statement: Else

Syntax: Else *[Statement...]*

Modes: Amiga/Blitz

Description:

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

Example:

```
;
; if...else...endif program example
;

NPrint "Type a number from 1 to 10"
a=Edit(3)

If a<5
 NPrint "Your number is less than 5"
Else
 NPrint "Your number is greater than or equal to 5"
EndIf

MouseWait
```

See Also:

If, EndIf

Statement: **While**

Syntax: While *Expression*

Modes: Amiga/Blitz

Description:

The **While** command is 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**.

Example:

```
;
; while...wend program example
;

While a<10
 NPrint a
 a+1
Wend

MouseWait
```

See Also:

Wend, Repeat

Statement: **Wend**

Syntax: Wend

Modes: Amiga/Blitz

Description:

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

See Also:

While

Statement: **Select**

Syntax: Select *Expression*

Modes: Amiga/Blitz

Description:

Select examines and 'remembers' the result of the specified *Expression*. Later in the program, **Case** may used to executed different sections of program code depending on this result. Here is an example of a typical **Select...Case...End Select** sequence:

```
Select Expression
Case 1
;execute this if expression evaluated to 1
Case 2
;execute this if expression evaluated to 2
.
.may have many more 'Case's...
.
Default
;execute this if expression did not match any of the cases.
End Select
```

Example:

```
;
; select...case program example
;

Print "Enter a number from 1 to 3:"
n=Edit(80)

Select n
 Case 1
  NPrint "One!"
 Case 2
  NPrint "Two!"
 Case 3
  NPrint "Three!"
 Default
  NPrint "That number was not 1, 2 or 3!"
End Select

MouseWait
```

See Also:

Case, Default, End Select

Statement: **Case**

Syntax: Case *Expression*

Modes: Amiga/Blitz

Description:

A **Case** is 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* specified 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**.

See Also:

Select, **Default**, **End Select**

Statement: **Default**

Syntax: Default

Modes: Amiga/Blitz

Description:

A **Default** statement may appear following a series of **Case** statements to cause a section of program code to be executed if NONE of the **Case** statements were satisfied.

See Also:

Select, **Case**, **End Select**

Statement: **End Select**

Syntax: End Select

Modes: Amiga/Blitz

Description:

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

See Also:

Select, **Case**, **Default**

Statement: **For**

Syntax: For *Var=Expression1* ***To*** *Expression2* ***[Step*** *Expression3]*

Modes: Amiga/Blitz

Description:

The **For** statement initializes 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 the For/Next loop's **Next**, ending the loop. If not, program flow continues on until the loop's **Next** is reached. At this point, the value specified in *Expression3* (the '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.

An interesting feature of For/Next loops is the ability to use the loop's index variable within the loop.
In order for a For/Next loop to count 'down' from one value to a lower value, a negative step number must be supplied.

Example:

```
;
; nested for...next loops program example
;

For a=1 To 3                ;start up a for next loop
  For b=3 To 1 Step -1      ;and another, 'inner' loop
    NPrint "a=",a," b=",b   ;show what's happening to the index variables.
  Next                      ;next for 'b' For/Next loop...
Next                        ;next for 'a' For/Next loop...

MouseWait
```

See Also:

Next, Step

Statement: **Next**

Syntax: Next *[Var[,Var...]]*

Modes: Amiga/Blitz

Description:

Next terminates a For/Next loop. Please refer to the **For** command for more information on For/Next loops.

See Also:

For, Step

Statement: **Repeat**

Syntax: Repeat

Modes: Amiga/Blitz

Description:

Repeat is 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.

Example:

```
;
; repeat...until program example
;

Repeat

 Print "Type a number (0 to quit):"
 n=Edit(80)

 If n/2=Int(n/2)
  NPrint n," is an even number"
 Else
  NPrint n," is an odd number"
 EndIf

Until n=0
```

See Also:

Until, Forever

Statement: **Until**

Syntax: Until *Expression*

Modes: Amiga/Blitz

Description:

Until is used to terminate a Repeat/Until loop. If *Expression* proves to be true (non 0), then program flow will continue from the command following **Until**. If *Expression* proves to be false (0), then program flow will go back to the corresponding **Repeat**, found further up the program.

See Also:

Repeat, Forever

Statement: **Forever**

Syntax: Forever

Modes: Amiga/Blitz

Description:

Forever may be used instead of **Until** to cause a Repeat/Until loop to NEVER exit.

Executing **Forever** is identical to executing '**Until** 0'.

See Also:

Repeat, Until

Statement: **Pop**

Syntax: Pop ***Gosub/ For/ Select/ If/ While/ Repeat***

Modes: Amiga/Blitz

Description:

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 the program. However, to achieve this Blitz 2 must be told that the relevant loop should be 'forgotten'. This is the purpose of **Pop**.

Actually, **Pop** is only necessary to prematurely terminate **Gosubs**, **Fors** and **Selects**. **If**, **While** and **Repeat** have been included for completeness.

Example:

```
;
; guessing game program example (pop example in here somewhere)
;

Repeat

 NPrint "Think of a number between 1 and 1000..."
 NPrint "I Shall try to guess it in ten goes!"

 l=0:h=1000

 For k=1 To 10
  n=Int((h-l)/2)+l

  Repeat
   Print "Is your number ",n,"? (y)es, (h)igher, (l)ower ?"
```

```
    a$=LCase$(Edit$(1))
   Until a$="y" OR a$="h" OR a$="l"

   Select a$
    Case "y"
     NPrint "Clever, aren't I ?"
     NPrint "I got it in ",k," guesses!"
     Pop Select:Pop For
     Goto right
    Case "l"
     h=n
    Case "h"
     l=n
    End Select

  Next

  NPrint "Huh??? You must have CHEATED!"

 right:
  Print "Another Game ? (y)es, (n)o ?"
  a$=LCase$(Edit$(1))

Until a$="n"
```

Statement: **SetInt**

Syntax: SetInt *Type*

Modes: Amiga/Blitz

Description:

SetInt is used to declare a section of program code as 'interupt' code. Before going further into the details of **SetInt**, let's have a quick look at what interupts are.

Often, when a computer program is running, an event of some importance takes place which must be processed immediately. This is done through interupts. When an interupt occurs, whatever program may be currently running is completely halted by the 68000. Then, a program known as an 'interupt handler' is started. Once the interupt handler has done it's work, the program which was originally interupted is restarted, without any knowledge of having been disturbed.

So what can cause an interupt? On the Amiga, there are 14 different types of possible interupts, each assigned it's own special number. These interupts are as follows:

Interupt	Cause of Interupt
0	Serial transmit buffer empty
1	Disk Block read/written
2	Software interupt
3	Cia ports interupt
4	Co-processor ('copper') interupt
5	Vetical 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 interupt

The most useful of these interupts is the vertical blank interupt. This interupt occurs every time an entire video frame has been fully displayed (about every sixtieth of a second), and is very useful for animation purposes. If a section of program code has been designated as a vertical blank interupt handler, then that section of code will be executed every sixtieth of a second.

Interupt handlers must perform their task as quickly as possible, especially in the case of vertical blank handlers which must NEVER take longer than one sixtieth of a second to execute.

Interupt handlers in Blitz 2 must NEVER access string variables or literal strings. In Blitz mode, this is the only restriction on interupt handlers. In Amiga mode, no blitter, Intuition or file i/o commands may be executed by interupt handlers.

To set up a section of code to be used as an interupt handler, you use the **SetInt** command followed by the actual interupt handler code. An **End SetInt** should follow the interupt code. The *Type* parameter specifies the type of interupt, from the above table, the interupt handler should be attached to. For example, **SetInt** 5 should be used for vertical blank interupt code.

More than one interupt handler may be attached to a particular type of interupt.

Example:

```
;
; vertical blank interrupt routine program example
;

SetInt 5                ;vertical blank handler follows.....
 a+1                    ;add one to 'a'
 Poke.w $dff180,a            ;this little poke will change background colour
End SetInt               ;end of interupt handler

MouseWait                 ;wait for mouseclick - handler still going!
```

See Also:

End SetInt, ClrInt

Statement: **End SetInt**

Syntax: End SetInt

Modes: Amiga/Blitz

Description:

End SetInt must appear after a **SetInt** to signify the end of a section of interupt handler code. Please refer to **SetInt** for more information of interupt handlers.

See Also:

SetInt, ClrInt

Statement: **ClrInt**

Syntax: ClrInt *Type*

Modes: Amiga/Blitz

Description:

ClrInt may be used to remove any interupt handlers currently attached to the specified interupt *Type*. The **SetInt** command is used to attached interupt handlers to particular interupts.

Example:

```
;
; end setint program example
;

SetInt 5                    ;interupt handler follows...
 a+1                        ;add one to 'a'
 Poke.w $dff180,a                ;set background colour
End SetInt                    ;end of handler

NPrint "Hit return..."          ;handler going till return is hit...
b=Edit(1)                     ;do an edit function
ClrInt 5                    ;turn of all type 5 interupt handlers
NPrint "Click Mouse button..."
MouseWait
```

See Also:

SetInt, End SetInt

Statement: **SetErr**

Syntax: SetErr

Modes: Amiga/Blitz

Description:

The **SetErr** command allows you to set up custom error handlers. Program code which appears after the **SetErr** command will be executed when any Blitz 2 runtime errors are caused. Custom error code should be ended by an **End SetErr**.

Example:

```
;
; error handler example program
;

SetErr                        ;install error handler
 NPrint "RUNTIME ERROR!"        ;this is our handler...
 NPrint "Click Mouse Button."
 MouseWait
 ErrFail
End SetErr                     ;end of error handler

Dim a(10)                      ;dim an array
For k=1 To 11                   ;going to cause an error!
 a(k)=k
NPrint a(k)
Next
```

See Also:

ClrErr, ErrFail

Statement: **End SetErr**

Syntax: End SetErr

Modes: Amiga/Blitz

Description:

End SetErr must appear following custom error handlers installed using **SetErr**. Plase refer to **SetErr** for more information on custom error handlers.

See Also:

SetErr, ClrErr, ErrFail

Statement: ClrErr

Syntax: ClrErr

Modes: Amiga/Blitz

Description:

ClrErr may be used to remove a custom error handler set up using **SetErr**.

See Also:

SetErr, ErrFail, ClrErr

Statement: ErrFail

Syntax: ErrFail

Modes: Amiga/Blitz

Description:

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 reported and transfer will be passed to direct mode.

See Also:

SetErr, ClrErr

Statement: VWait

Syntax: VWait *[Frames]*

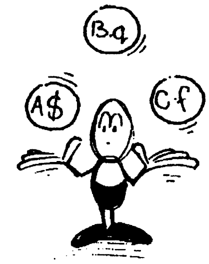
Modes: Amiga/Blitz

Description:

VWait will cause program flow to halt until the next vertical blank occurs. The optional *Frames* parameter may be used to wait for a particular number of vertical blanks.

VWait is especially useful in animation for synchronizing display changes with the rate at which the display is physically redrawn by the monitor.

2. Variable Handling



This section covers all commands related to Blitz 2 variable handling. This includes the handling of standard types as well as Blitz 2's NewTypes, arrays, lists, and data statements. NewTypes are Blitz's answer to C structures while Lists refer to Blitz's linked list capabilities including a whole command set supporting all standard operations on linked lists.

Statement: **Let**

Syntax: Let *Var=l Operator Expression*

Modes: Amiga/Blitz

Description:

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. Normally, an equals sign ('=') is placed between the variable name and the expression. If the equals sign is omitted, then an opertor (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. Here are some examples of **Let**:

Example:

```
;
; let program example
;

Let a=10          ;assign 10 to 'a'
Let a=b*5          ;assign 'b times 5' to 'a'
Let k+1           ;add 1 to 'k'
Let z*5           ;multiply 'z' by 5.
```

Statement: **Data**

Syntax: Data *[.Type] Item[,Item...]*

Modes: Amiga/Blitz

Description:

The **Data** statement allows you to include pre-defined values in your programs. These '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.

Example:

```
;
; read data program example
;

Read a$,b,c.w              ;read next 3 pieces of data.
NPrint a$               ;print them out...
NPrint b
NPrint c
MouseWait
End

Data$ "Some data to be read"   ;data to be read - string...
Data 10                        ;quick...
Data.w -5                      ;and word.
```

See Also:

Read, Restore

Statement: **Read**

Syntax: Read *Var[,Var...]*

Modes: Amiga/Blitz

Description:

Read is used to transfer items in **Data** statements into variables. Data is transferred sequentially into variables through what is known as a 'data pointer'. Each time a pice of data is read, the data pointer is incremented to point at the next piece of data. The data pointer may be set to point to a particular piece of data using the **Restore** command.

See Also:

Data, Restore

Statement: **Restore**

Syntax: Restore *[Program Label]*

Modes: Amiga/Blitz

Description:

Restore allows you to set Blitz 2's internal 'data pointer' to a particular piece of data. after executing a **Restore,** The first item of data following the specified *Program Label* will become the data to be read when the next **Read** command is executed.

Restore with no parameters will reset the data pointer to the very first piece of data in the program.

See Also:

Data, Read

Statement: **Exchange**

Syntax: Exchange *Var,Var*

Modes: Amiga/Blitz

Description:

Exchange will 'swap' the values contained in the 2 specified variables. **Exchange** may only be used with 2 variables of the same type.

Example:

```
;
; exchange program example
;

a=10            ;put 10 into 'a'
b=20            ;put 20 into 'b'
NPrint a        ;print a & b
NPrint b
Exchange a,b        ;exchange variables...
NPrint a        ;print a & b again...
NPrint b
MouseWait
```

Statement: **MaxLen**

Syntax: MaxLen *StringVar=Expression*

Modes: Amiga/Blitz

Description:

MaxLen sets aside a block of memory for a string variable to grow into. This is normally only necessary in the case of special Blitz 2 commands which require this space to be present before execution. Currently, only 2 Blitz 2 commands require the use of **MaxLen** - **FileRequest$** and **Fields**.

Example

```
;
; filerequest program example
;

WbToScreen 0        ;pick up workbench as currently used screen
WBenchToFront_       ;bring workbench to front of view
MaxLen pa$=160       ;these are necessary for FileRequest$...
MaxLen fi$=64       ;to operate properly!
```

```
a$=FileRequest$("Select a File",pa$,fi$)  ;bring up a file requester

WBenchToBack_          ;workbench back to rear of view.
```

See Also:

FileRequest$, Fields

Statement: DEFTYPE

Syntax: DEFTYPE *.Typename [Var[,Var...]]*

Modes: Amiga/Blitz

Description:

DEFTYPE may be used in 2 ways:

* **DEFTYPE** may be used to declare a list of variables as being of a particular type. In this case, *Var* parameters must be supplied.

* **DEFTYPE** may 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.

Please refer to the Programming chapter of the Blitz 2 Programmers guide for more information on variable types and the use of **DEFTYPE**.

Example:

```
;
; deftype program example
;
DEFTYPE.l a,b,c      ;these variables are all 'longs'
d=10                 ;'d' is a quick (the initial default type)
DEFTYPE.w            ;set default type to 'word'
e=10                 ;'e' is a word
```

See Also:

NEWTYPE

Statement: NEWTYPE

Syntax: NEWTYPE *.Typename*

Modes: Amiga/Blitz

Description:

NEWTYPE is used to create a custom variable type. **NEWTYPE** must be followed by a list of entry names separated by colons (':') and/or newlines. **NEWTYPE**s are terminated using **End NEWTYPE**. Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on setting up and using custom variable types.

Example:

```
;
; newtype program example
;

NEWTYPE.test          ;start of custom variable type.
 a.l                ;contents of type...
 b.w                 ;...
 c.q                 ;...
End NEWTYPE           ;end of custom variable type.

a.test\a=10,20,30      ;assign some values.

NPrint a\a,a\b,a\c     ;output values

MouseWait
```

See Also:

DEFTYPE, USEPATH

Function: **SizeOf**

Syntax: SizeOf .*Typename[,Entrypath]*

Modes: Amiga/Blitz

Description:

SizeOf allows you to determine the amount of memory, in bytes, a particular variable type takes up. **SizeOf** 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.

Example:

```
;
; sizeof program example
;

NEWTYPE.test          ;create a custom variable type...
 a.l
 b.w
 c.q
End NEWTYPE           ;end of custom variable type.

NPrint SizeOf.b     ;print size of a byte!

NPrint SizeOf.test    ;print size of our custom type

NPrint SizeOf.test\b   ;print offset to 'b' entry of our type.
```

```
MouseWait
```

See Also:

NEWTYPE

Statement: Dim

Syntax: Dim *Arrayname [**List**] (Dimension1[,Dimension2...])*

Modes: Amiga/Blitz

Description:

Dim is used to initialize a BASIC array. Blitz 2 supports 2 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 1 dimension.

Lists are covered fully in the Blitz 2 programmers guide, under the programming section.

Example:

```
;
; array example
;

Dim a(3,3)                    ;initialize 'a' array

For k=1 To 3                  ;outer loop...
 For j=1 To 3                 ;inner loop...
  a(k,j)=c                    ;assign array element
  c+1                         ;increment 'c'
 Next                         ;end of inner loop
Next                          ;end of outer loop

For k=1 To 3                  ;outer loop...
 For j=1 To 3                 ;inner loop...
  NPrint "a(",k,",",j,")=",a(k,j)     ;print out array elements
 Next                         ;end of inner loop
Next

MouseWait
```

Statement: ResetList

Syntax: ResetList *Arrayname()*

Modes: Amiga/Blitz

Description:

ResetList is 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 item.

Example:

```
;
; list program example
;

Dim List a(10)              ;initialize a list array...

While AddFirst(a())         ;fill it up with stuff
 a()=c
 c+1
Wend

NPrint "Contents of a()..."

ResetList a()               ;back to first item in list

While NextItem(a())         ;process list
 NPrint a()                 ;output value of element
Wend

MouseWait
```

See Also:

NextItem

Statement: **ClearList**

Syntax: ClearList *Arrayname()*

Modes: Amiga/Blitz

Description:

ClearList is used in conjunction with list arrays to completely 'empty' out the specified list array. List arrays are automatically emptied when they are **Dim**med.

See Also:

Dim, ResetList

Function: **AddFirst**

Syntax: AddFirst *(Arrayname())*

Modes: Amiga/Blitz

Description:

The **AddFirst** function allows you to insert an array list item at the beginning of an array list. **AddFirst** 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).

Example:

```
;
; addfirst program example
;

Dim List a(100)              ;initialize list array

While AddFirst(a())             ;while an item is available...
  a()=c                      ;set it to something...
  c+1                        ;increment counter
Wend

NPrint c," items successfully added."   ;output how many items added

MouseWait
```

See Also:

AddLast, AddItem, Killitem

Function: **AddLast**

Syntax: AddLast *(Arrayname())*

Modes: Amiga/Blitz

Description:

The **AddLast** function 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).

See Also:

AddFirst, AddItem, KillItem

Function: **AddItem**

Syntax: AddItem *(Arrayname())*

Modes: Amiga/Blitz

Description:

The **AddItem** function 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).

Example:

```
;
; list handling program example
;

Dim List a(10)

If AddFirst(a()) Then a()=1

If AddItem(a()) Then a()=2

NPrint "List Array (first to last) is..."

ResetList a()

While NextItem(a())
  NPrint a()
Wend

MouseWait
```

See Also:

AddFirst, AddLast, KillItem

Statement: **KillItem**

Syntax: KillItem *ArrayName()*

Modes: Amiga/Blitz

Description:

KillItem is used to delete 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.

Example:

```
;
; process list with killitem program example
;

Dim List a(10)            ;initialize list array

While AddItem(a())           ;fill list...
  a()=c                   ;with sequential values...
  c+1
Wend

ResetList a()             ;reset list...
```

```
While NextItem(a())         ;process list...
 If a()/2<>Int(a()/2)       ;is item odd ?
  KillItem a()              ;yes, kill it!
 EndIf
Wend

NPrint "Final List (Odd elements deleted) is..."

ResetList a()               ;reset list

While NextItem(a())         ;output all elements...
 NPrint a()
Wend

MouseWait
```

See Also:

AddFirst, AddLast, AddItem

Function: **PrevItem**

Syntax: PrevItem *(Arrayname())*

Modes: Amiga/Blitz

Description:

PrevItem 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).

Example:

```
;
; print list backwards program example
;

Dim List a(10)              ;initialize list array

While AddLast(a())          ;fill list...
 a()=c                      ;with 0,1,2...
 c+1
Wend

NPrint "List contents (backwards) are..."

If LastItem(a())            ;go to last item in list
 Repeat                     ;repeat...
  NPrint a()
 Until NOT PrevItem(a())    ;until no more previous items
EndIf
```

```
MouseWait
```

See Also:

Nextitem

Function: **NextItem**

Syntax: NextItem *(Arrayname())*

Modes: Amiga/Blitz

Description:

NextItem will set the specified list array's 'current item' pointer to the item *after* the list array's old current item. This allows for 'forwards' 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, **NextItem** will return false (0).

Example:

```
;
; print list forwards program example
;

Dim List a(10)              ;initialize list array

While AddLast(a())              ;fill list
 a()=c                      ;with stuff...
 c+1
Wend

NPrint "List contents (forwards) are..."

ResetList a()               ;reset list

While NextItem(a())             ;output items in list...
 NPrint a()
Wend

MouseWait
```

See Also:

PrevItem

Function: **FirstItem**

Syntax: FirstItem *(Arrayname())*

Modes: Amiga/Blitz

Description:

Executing **FirstItem** 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).

Example:

```
;
; print lastitem in list
;

Dim List a(10)          ;initialize list array

While AddFirst(a())     ;fill list array...
  a()=c
  c+1
Wend

If FirstItem(a())                ;if there is a lastitem...
  NPrint "First Item in list is:",a()  ;print it out...
EndIf

MouseWait
```

See Also:

LastItem

Function: **LastItem**

Syntax: LastItem *(Arrayname())*

Modes: Amiga/Blitz

Description:

Executing **LastItem** 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).

Example:

```
;
; print lastitem in list
;

Dim List a(10)          ;initialize list array

While AddLast(a())      ;fill list array...
  a()=c
  c+1
Wend
```

```
If LastItem(a())              ;if there is a lastitem...
  NPrint "Last Item in list is:",a()  ;print it out...
EndIf

MouseWait
```

See Also:

FirstItem

Statement: PushItem

Syntax: PushItem *Arrayname()*

Modes: Amiga/Blitz

Description:

Executing **PushItem** 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 allows for up to 8 'pushes'.

Example:

```
;
; pushing items on stack with list
;

Dim List a(10)               ;initialize list array

While AddLast(a())             ;fill array up with 0...9
  a()=c
  c+1
Wend

ResetList a()                ;reset list

While NextItem(a())            ;process all items
  If a()=5 Then PushItem a()     ;remember when '5' found
Wend

PopItem a()                  ;recall '5'
KillItem a()                 ;delete it.

ResetList a()                ;reset list

While NextItem(a())            ;output list contents
  NPrint a()
Wend

MouseWait

End
```

See Also:

PopItem

Statement: **PopItem**

Syntax: PopItem *Arrayname()*

Modes: Amiga/Blitz

Description:

PopItem 'pops' or 'recalls' a previously pushed current item pointer for the specified list array. *Arrayname()* must match the arrayname of the most recently executed **PushItem**.

See Also:

PushItem

Statement: **ItemStackSize**

Syntax: ItemStackSize *Max Items*

Modes: Amiga/Blitz

Description:

ItemStackSize determines how many 'list' items may be pushed (using the **PushItem** command), before items must be '**Pop**'ped 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.

See Also:

PushItem, **Popitem**

Statement: **Sort**

Syntax: Sort *Arrayname()*

Modes: Amiga/Blitz

Description:

Sort will cause the specified array to be sorted. Only primitive type, 'non-list' arrays may be sorted; it is not possible to sort newtype arrays, or 'list' arrays.

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.

Example:

```
;
; a sort of an example
;
Dim a(9)           ;dimension an 'a' array
For k=0 To 9       ;fill array with random values...
  a(k)=Rnd
Next
Sort a()           ;sort the array
For k=0 To 9       ;print out sorted array
  NPrint a(k)
Next
MouseWait          ;wait for mouse click
```

See Also:

SortUp, SortDown

Statement: **SortUp**

Syntax: SortUp

Modes: Amiga/Blitz

Description:

SortUp may be used to force the **Sort** command to sort arrays into ascending order. This means that, after being sorted, an array's contents will be ordered in a 'low to high' manner.

See Also:

Sort, SortDown

Statement: **SortDown**

Syntax: SortDown

Modes: Amiga/Blitz

Description:

SortDown may be used to force the **Sort** command to sort arrays into descending order. This means that, after being sorted, an array's contents will be ordered in a 'high to low' manner.

See Also:

Sort, SortUp

3. Procedures

This section covers the commands related to Statements and Functions in Blitz 2. Local and global variables as well as recursion are all discussed in detail.

Statement: **Statement**

Syntax: Statement *Procedurename{[Parameter1[,Paramater2...]]}*

Modes: Amiga/Blitz

Description:

Statement 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 2, all statements and functions must be declared before they are called.

Example:

```
;
; declare a statement program example
;

Statement hexprint{a}  ;declare statement with one parameter
  NPrint Hex$(a)       ;print out hex value of parameter
End Statement          ;end of statement

hexprint{16384}        ;call statement

MouseWait
```

See Also:

End Statement, Statement Return, Function

Statement: **End Statement**

Syntax: End Statement

Modes: Amiga/Blitz

Description:

End Statement declares the end of a 'statement type' procedure definition. All statement type procedures must be terminated with an **End Statement**.

See Also:

Statement, Statement Return

Statement: **Statement Return**

Syntax: Statement Return

Modes: Amiga/Blitz

Description:

Statement Return may be used to prematurely exit from a 'statement type' procedure. Program flow will return to the command following the procedure call.

Example:

```
;
; statement variable passing program example
;

Statement printeven{a}                ;start of procedure
  If a/2<>Int(a/2) Then Statement Return  ;if parameter is odd, exit.
  NPrint a                    ;else print parameter
End Statement                 ;end of procedure

For k=1 To 10                 ;start of loop
 printeven{a}                 ;call statement
Next                          ;end of loop

MouseWait
```

See Also:

End Statement, Function Return

Statement: **Function**

Syntax: Function *[.Type] Procedurename{[Parameter1[,Parameter2...]]}*

Modes: Amiga/Blitz

Description:

Function 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 2'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 the form of local variables through which calling parameters are passed.

Functions may return values through the **Function Return** command.

In Blitz 2, all statements and functions must be declared before they are called.

Example:

```
;
; function program example
;

Function$ hexof{a}          ;declare function with one parameter
  Function Return Hex$(a)     ;return hex value of parameter
End Function                ;end of function

NPrint hexof{16384}         ;call function

MouseWait
```

See Also:

End Function, Function Return, Statement

Statement: **End Function**

Syntax: End Function

Modes: Amiga/Blitz

Description:

End Function declares the end of a 'function type' procedure definition. All function type procedures must be terminated with an **End Function**.

See Also:

Function, Function Return

Statement: **Function Return**

Syntax: Function Return *Expression*

Modes: Amiga/Blitz

Description:

Function Return allows 'function type' procedures to return values to their calling expressions. Function type procedures are always called from within Blitz 2 expressions.

Example:

```
;
; function example
;

Function double{a}      ;start of function code...
 Function Return a+a  ;return double the passed parameter
End Function           ;end of function code.

For k=1 To 10          ;start of loop
 NPrint double{k}     ;output 'k' doubled
Next                   ;end of loop

MouseWait
```

See Also:

End Function, Statement Return

Statement: **Shared**

Syntax: Shared *Var[,Var...]*

Modes: Amiga/Blitz

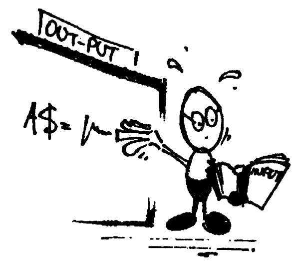
Description:

Shared is 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.

Example:

```
;
; local variable program example
;

Statement test{a}        ;start of procedure definition
 Shared k              ;use global 'k' variable
 NPrint k*a            ;output 'k' times parameter
End Statement            ;end of procedure definition

For k=1 To 10            ;start of loop
 NPrint test{5}         ;call 'test
Next                     ;end of loop

MouseWait
```

4. Input/Output



The following section details Blitz 2's BASIC input/output commands including the print and edit commands as well as joystick input, print formatting and default input and output redirection.

Statement: **Print**

Syntax: Print *Expression[,Expresion...]*

Modes: Amiga/Blitz

Description:

Print 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.

Example:

```
;
; print program example
;
Print "Hello "
Print "There! "
a=2
Print "Blitz Basic ",a," at work!"
MouseWait
```

See Also:

NPrint

Statement: **NPrint**

Syntax: NPrint *Expression[,Expresion...]*

Modes: Amiga/Blitz

Description:

NPrint 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.

Example:

```
;
; nprint program example
;
NPrint "Hello "
NPrint "There!"
a=2
NPrint "Blitz Basic ",a," at work!"
NPrint "Goodbye..."
MouseWait
```

See Also:

Print

Statement: **Format**

Syntax: Format *FormatString*

Modes: Amiga/Blitz

Description:

Format 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:

Character	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.

Here are some example of *FormatStrings* and their output:

FormatString	Value printedOutput
"####.00"	5.2 5.20
"0000.00"	5.20005.20
"###,###.00"	10240.25 10,240.25
"Total: -#####"	-10.5Total: - 11

Format affects the operation of the **Str$** function.

See Also:

Str$

Statement: **FloatMode**

Syntax: FloatMode *Mode*

Modes: Amiga/Blitz

Description:

FloatMode 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 - in exponential format, or in standard format. Exponential format displays a floating point number as a value multiplied by ten raised to a power. For example, 10240 expressed exponentially is displayed as '1.024E+4', ie: 1.024 times 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 floating point values to ALWAYS be displayed in standard format.
A *Mode* parameter of 0 will cause Blitz 2 to take a 'best guess' at the most appropriate format to use. This is the default mode for floating point output.

Note that if **Format** has been used to alter numeric output, standard mode will always be used to print floating point numbers.

Example:

```
;
;floatmode program example
;
a.f=10240.25
NPrint a
FloatMode 1
NPrint a
FloatMode -1
NPrint a
MouseWait
```

Function: **Joyx**

Syntax: Joyx *(Port)*

Modes: Amiga/Blitz

Description:

Joyx will return the left/right status of a joystick plugged into the specified port. Port must be either 0 or 1, 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.

See Also:

Joyy, Joyr, Joyb

Function: Joyy

Syntax: Joyy *(Port)*

Modes: Amiga/Blitz

Description:

Joyy will return the up/down status of a joystick plugged into the specified port. Port must be either 0 or 1, 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 downwrads, **Joyy** will return 1. If the joystick is held neither upwards or downwards, **Joyy** will return 0.

See Also:

Joyx, Joyr, Joyb

Function: Joyr

Syntax: Joyr *(Port)*

Modes: Amiga/Blitz

Description:

Joyr may be used to determine the rotational direction of a joystick plugged into the specified port. *Port* must be either 0 or 1, port 0 beng the port the mouse is normally plugged into.

Joyr returns a value from 0 through 8 based on the following table:

Joystick direction	Joyr value
Up	0
Up-Right	1
Right	2
Down-Right	3
Down	4
Down-Left	5
Left	6
Up-Left	7
No Direction	8

See Also:

Joyx, Joyy, Joyb

Function: **Joyb**

Syntax: Joyb *(Port)*

Modes: Amiga/Blitz

Description:

Joyb allows you to read the button status of the device plugged into the specified port. *Port* must be either 0 or 1, 0 being the port the mouse is normally plugged into.

If the left button is held down, **Joyb** will return 1. If the right button is held down, **Joyb** will return 2. If both buttons are held down, **Joyb** will return 3. If no buttons are held down, **Joyb** will return 0.

See Also:

Joyx, Joyy, Joyr

Statement: **DefaultInput**

Syntax: DefaultInput

Modes: Amiga/Blitz

Description:

DefaultInput causes all future **Edit$** functions to receive their input from the CLI window the Blitz 2 program was run from. This is the default input channel used when a Blitz 2 program is first run.

See Also:

DefaultOutput

Statement: **DefaultOutput**

Syntax: DefaultOutput

Modes: Amiga/Blitz

Description:

DefaultOutput cause all future **Print** statements to send their output to the CLI window the Blitz 2 program was run from. This is the default output channel used when a Blitz 2 program is first run.

See Also:

DefaultInput

Function: **FileRequest$**

Syntax: FileRequest$ *(Title$,Pathname$,Filename$)*

Modes: Amiga

Description:

The **FileRequest$** function will open up a standard Amiga-style file requester on the currently used screen. Program flow will halt until the user either selects a file, or hits the requester's 'Cancel' button. If a file was selected, **FileRequest$** will return the full 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.

Pathname$ MUST be a string variable with a **MaxLen** of at least 160.

Filename$ MUST be a string variable with a **MaxLen** of at least 64.

Example:

```
;
; file request example program
;
WbToScreen 0                 ;use workbench
WBenchToFront_                ;workbench to front
MaxLen pa$=160                 ;set 'path' string var
MaxLen fi$=64                ;set 'file' string var
a$=FileRequest$("Select a File",pa$,fi$) ;do file requester
WBenchToBack_                  ;Workbench to back
```

See Also:

MaxLen

Function: **Edit$**

Syntax: Edit$ *([DefaultString$],Characters)*

Modes: Amiga/Blitz

Description:

Edit$** is Blitz 2's standard text input command. **Edit$ normally causes the following chain of events:

* The optional *DefaultString$* and a cursor is printed to the display.

* The program user types in a string of text.

* When 'RETURN' is hit, **Edit$** returns the text entered by the program user as a string of character.

Edit$ operates slightly differently depending on the mode of input at the time of execution. For instance, executing a **WindowInput** command will cause **Edit$** to receive and print it's input to an Intuition window, whereas executing **FileInput** will cause **Edit$** to receive it's input from a file.

Characters specifies a maximum number of allowable characters for input. This is extremely useful in preventing **Edit$** from destroying display contents.

Example:

```
;
; edit$ program example
;
Print "Please Type in your name:"     ;prompt for a name
a$=Edit$(40)                          ;receive input
NPrint "Hello There ",a$," !"         ;print message and name
MouseWait
```

See Also:

Edit, Inkey$

Function: **Edit**

Syntax: Edit *([DefaultValue],Characters)*

Modes: Amiga/Blitz

Description:

Edit is Blitz 2's standard numeric input command. **Edit** normally causes the following chain of events:

* The optional *Defaultvalue* and a cursor is printed to the display.

* The program user types in a numeric value.

* When 'RETURN' is hit, **Edit** returns the value entered by the program user.

Edit operates slightly differently depending on the mode of input at the time of execution. For instance, executing a **WindowInput** command will cause **Edit** to receive and print it's input to an Intuition window, whereas executing **FileInput** will cause **Edit** to receive it's input from a file.

Characters specifies a maximum number of allowable characters for input. This is extremely useful in preventing **Edit** from destroying display contents.

Example:

```
;
; edit program example
;
Print "Type in your age:"             ;prompt...
a=Edit(40)                            ;receive age!

If a>=21                              ;are they over 21?
  NPrint "I hope you enjoyed your twenty first!"         ;yes!
Else                                                     ;else...
 NPrint "I bet you're looking forward to your twenty first!" ;no!
EndIf

MouseWait
```

See Also:

Edit$, Inkey$

Function: **Inkey$**

Syntax: Inkey$ *[(Characters)]*

Modes: Amiga/Blitz

Description:

Inkey$ may be used to collect one or more characters from the current input channel. The current input channel may be selected using commands such as **WindowInput**, **FileInput** or **BitMapInput**. **Inkey$** MAY NOT be used from the **DefaultInput** input channel.

Characters refers to the number of characters to collect. The default is one character.

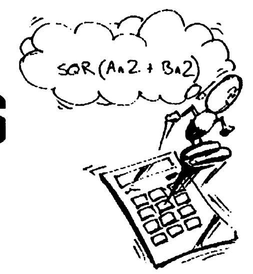
Example:

```
;
; inkey$ program example
;
Screen 0,3
Window 0,0,0,320,200,$100f,"My Window!",1,2
NPrint "Type away - hit Mouse Button to Quit!"

While Joyb(0)=0   ;loop continuously until a mousebutton down
 WaitEvent
 Print Inkey$
Wend
```

See Also:

Edit$, Edit

5. Numeric Functions



This section covers all functions which accept and return only numeric values. Note that all the transcendental functions (eg. **Sin**, **Cos**) operate in radians.

Function: **NTSC**

Syntax: **NTSC**

Modes: Amiga/Blitz

Description:

This function 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.

Example:

```
;
; NTSC test example program
;
If NTSC Then Print "Yo Dude" Else Print "Hello Chaps"
MouseWait
```

See Also:

DispHeight

Function: **DispHeight**

Syntax: DispHeight

Modes: Amiga/Blitz

Description:

DispHeight will return 256 if executed on a PAL Amiga, or 200 if executed on an NTSC Amiga. This allows programs to open full sized screens, windows etc on any Amiga.

Example:

```
;
; max display height example program
;
Print "Maximum display height is ",DispHeight
MouseWait
```

See Also:

NTSC

Function: **Peek**

Syntax: **Peek** *[.Type](Address)*

Modes: Amiga/Blitz

Description:

Peek returns the value found at the memory location specified by *Address*. The value returned depends on the size of the peek. If **Peek.b** is used the byte at memory location *MemLoc* is returned.

If **Peek.w** is used the word at memory location *MemLoc* is returned. And for **Peek.l** or **Peek.q** the long word of the memory location is returned.

Peek$ may be used to read a null terminated string from memory.

Example:

```
;
; peek example program
;
NPrint "Exec Base can be found at"
Print Peek.l (4)
MouseWait
```

See Also:

Poke

Function: **Abs**

Syntax: **Abs** *(Expression)*

Modes: Amiga/Blitz

Description:

This function returns the positive equivalent of *Expression.*

Example:

```
Print Abs(-23) ; Prints 23 too
```

See Also:

QAbs

Function: **Frac**

Syntax: **Frac** *(Expression)*

Modes: Amiga/Blitz

Description:

Frac() returns the fractional part of *Expression.*

Example:

```
Print Frac(23.456) ; Will print .456
```

See Also:

QFrac

Function: **Int**

Syntax: **Int** *(Expression)*

Modes: Amiga/Blitz

Description:

This returns the Integer part (before the decimal point) of *Expression.*

Example:

```
Print Int(23.456)     ; Will simply print 23
```

Function: **QAbs**

Syntax: **QAbs** *(Quick)*

Modes: Amiga/Blitz

Description:

QAbs 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.

Example:

```
Print QAbs(-23) ; Prints 23
```

See Also:

Abs

Function: **QFrac**

Syntax: **QFrac** *(Quick)*

Modes: Amiga/Blitz

Description:

QFrac() returns the fractional part of a quick value. It works like **Frac()** but accepts a quick value as it's argument. It is faster than **Frac()** but has the normal quick value limits.

Example:

Print QFrac(23.4) ; *Prints .4*

See Also:

Frac

Function: **QLimit**

Syntax: **QLimit** *(Quick,Low,High)*

Modes: Amiga/Blitz

Description:

QLimit is used to limit the range of a quick number. If *Quick* is greater than or equal to *Low*, and less 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.

Example:

Print QLimit(150,0,140) ; *Prints 140*

Print QLimit(75,90,200) ; *Prints 90*

See Also:

QWrap

Function: **QWrap**

Syntax: **QWrap** *(Quick,Low,High)*

Modes: Amiga/Blitz

Description:

QWrap 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.

Example:

Print QWrap(-5,0,320) ; *Prints 315*

Print QWrap(325,0,320) ; *Prints 5*

See Also:

QLimit

Function: **Rnd**

Syntax: **Rnd** *[(Range)]*

Modes: Amiga/Blitz

Description:

This function returns a random number. If *Range* is not 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.

Example:

```
;
; random numbers program example
;
Screen 0,0,0,320,200,2,0,"1000 RANDOM PLOTS",1,2
ScreensBitMap 0,0
BitMapOutput 0
;
For i=1 To 1000
 Plot Rnd(320),Rnd(200),1   ;generate random numbers for x & y
Next
;
MouseWait
```

Function: **Sgn**

Syntax: **Sgn** *(Expression)*

Modes: Amiga/Blitz

Description:

Sgn returns the sign of *Expression.* If *Expression* is less than 0, then -1 is returned. If *Expression* is equal to 0, then 0 is returned. If *Expression* is greater than 0, then 1 is returned.

Example:

```
Print Sgn(-23)  ; Prints -1
Print Sgn(0)    ; Prints 0
Print Sgn(123)  ; Prints 1
```

Function: Cos

Syntax: **Cos** *(Float)*

Modes: Amiga/Blitz

Description:

Cos() returns the Cosine of the value *Float.*

Example:

```
;
; cosine curve program example
;
Screen 0,0,0,320,200,2,0,"A COSINE CURVE",1,2
ScreensBitMap 0,0
BitMapOutput 0
Locate 0,2:Print " 1"
Locate 0,12:Print " 0"
Locate 0,22:Print "-1"
Locate 19,13:Print "Pi"
Locate 37,13:Print "2Pi"
;
Line 16,20,16,180,2
Line 16,100,319,100,2
;
For k.f=0 To 1 Step .0025
;
Plot k*303+16,Cos(Pi*2*k)*80+100,3
;
Next
;
MouseWait
```

Function: Sin

Syntax: **Sin** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the Sine of the value *Float.*

Example:

```
;
; sine curve program example
;
Screen 0,0,0,320,200,2,0,"A SINE CURVE",1,2
ScreensBitMap 0,0
BitMapOutput 0
;
```

```
Locate 0,2:Print " 1"
Locate 0,12:Print " 0"
Locate 0,22:Print "-1"
Locate 19,13:Print "Pi"
Locate 37,13:Print "2Pi"
;
Line 16,20,16,180,2
Line 16,100,319,100,2
;
For k.f=0 To 1 Step .0025
;
Plot k*303+16,Sin(Pi*2*k)*80+100,3
;
Next
;
MouseWait
```

Function: **Tan**

Syntax: **Tan** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the Tangent of the value *Float*.

Example:

```
;
;
; tangent function program example
;
; for this to work, you'll have to turn off overflow
; checking from the runtime errors requester!
;
Screen 0,0,0,320,200,2,0,"A TAN CURVE",1,2
ScreensBitMap 0,0
BitMapOutput 0
;
Locate 0,2:Print " 10"
Locate 0,12:Print " 0"
Locate 0,22:Print "-10"
Locate 19,13:Print "Pi"
Locate 37,13:Print "2Pi"
;
Line 16,20,16,180,2
Line 16,100,319,100,2
;
For k.f=0 To 1 Step .0025
;
Plot k*303+16,Tan(Pi*2*k)*8+100,3
;
Next
```

MouseWait

Function: ACos

Syntax: **ACos** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the ArcCosine of the value *Float*.

Function: ASin

Syntax: **ASin** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the ArcSine of the value *Float*.

Function: ATan

Syntax: **ATan** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the ArcTangent of the value *Float*.

Function: HCos

Syntax: **HCos** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the hyperbolic Cosine of the value *Float*.

Function: **HSin**

Syntax: **HSin** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the hyperbolic Sine of the value *Float*.

Function: **HTan**

Syntax: **HTan** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the hyperbolic Tangent of the value *Float*.

Function: **Exp**

Syntax: **Exp** *(Float)*

Modes: Amiga/Blitz

Description:

This returns e raised to the power of *Float*.

Function: **Sqr**

Syntax: **Sqr** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the square root of *Float*.

Example:

```
;
; square root program example
;
opp=20
adj=50
hypot=Sqr(opp^2+adj^2)  ;Mr. Pythagoras' Rule
Print hypot
MouseWait
```

Function: **Log10**

Syntax: **Log10** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the base 10 logarithm of *Float*.

Function: **Log**

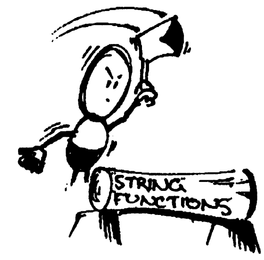
Syntax: **Log** *(Float)*

Modes: Amiga/Blitz

Description:

This returns the natural (base e) logarithm of *Float*.

6. String Functions



String functions include any functions which either return or accept a string expression.

Function: **Left$**

Syntax: **Left$** *(String$,Length)*

Modes: Amiga/Blitz

Description:

This function returns *Length* leftmost characters of string *String$.*

Example:

```
Print Left$("Hello there.",5): ; Will only print Hello
```

See Also:

UnLeft$**, **Right$

Function: **Right$**

Syntax: **Right$** *(String$,Length)*

Modes: Amiga/Blitz

Description:

Right$() returns the rightmost *Length* characters from string *String$.*

Example:

```
Print Right$("Hello there",5): ; Will just print there
```

See Also:

UnRight$**, **Left$

Function: **Mid$**

Syntax: **Mid$(***String$,Startchar[,Length]***)**

Modes: Amiga/Blitz

Description:

This function 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.

Example:

Print Mid$("Hello there",4,5): ; *Will Print the characters "lo th"*

Function: **Hex$**

Syntax: **Hex$** *(Expression)*

Modes: Amiga/Blitz

Description:

Hex$() returns an 8 character string equivalent to the hexadecimal representation of *Expression.*

Example:

Print Hex$(32): ; *Will print the string 00000020*

See Also:

Bin$

Function: **Bin$**

Syntax: **Bin$** *(Expression)*

Modes: Amiga/Blitz

Description:

Hex$() returns a 32 character string equivalent to the binary representation of *Expression.*

Example:

Print Bin$(32): ; *Will print 00000000000000000000000000100000*

See Also:

Hex$

Function: **Chr$**

Syntax: **Chr$** *(Expression)*

Modes: Amiga/Blitz

Description:

Chr$ 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.

Example:

Print Chr$(65): ; *Will print the letter A*

See Also:

Asc

Function: **Asc**

Syntax: **Asc** *(String$)*

Modes: Amiga/Blitz

Description:

Asc() returns the ASCII value of the first characters in the string *String$.*

Example:

Print Asc("A"): ; *Will print the number 65*

See Also:

Chr$

Function: **String$**

Syntax: **String$(***String$,Repeats***)**

Modes: Amiga/Blitz

Description:

This function will return a string containing *Repeats* sequential occurrences of the string *String$.*

Example:

Print String$("Hi!",3): ; *Will print Hi!Hi!Hi!*

Function: **Instr**

Syntax: **Instr** *(String$,Findstring$[,Startpos])*

Modes: Amiga/Blitz

Description:

Instr attempts to locate *FindString$* within *String$*. If a match is found, the character position of the first matching character will be returned. If no match is found, 0 will be returned.

The optional *Startpos* parameter allows you to specify a starting character position for the search.

CaseSense may be used to determine whether the search is case sensitive or not.

Example:

Print Instr("Hello there all","all"): ; *Will print 13*

Print Instr("Hello Hello","Hello",2): ; *Will print 7*

See Also:

CaseSense

Function: **Replace$**

Syntax: **Replace$** *(String$,Findstring$,Replacestring$)*

Modes: Amiga/Blitz

Description:

Replace$() will search the string *String$* for any occurrences of the string *Findstring$* and replace it with the string *Replacestring$*.

CaseSense may be used to determine whether the search is case sensitive or not.

Example:

Print Replace$("a a a ","a ","b-"): ; *Will print b-b-b-*

See Also:

CaseSense

Function: **Mki$**

Syntax: **Mki$** *(Integer)*

Modes: Amiga/Blitz

Description:

This will create a two byte character string, given the two byte numeric value *Numeric*.

Mki$ is often used before writing integer values to sequential files to save on disk space. When the file is later read in, **Cvi** may be used to convert the string back to an integer.

Example:

Print Mki$($4141): *; Prints "AA"*

See Also:

Cvi

Function: Mkl$

Syntax: **Mkl$** *(Long)*

Modes: Amiga/Blitz

Description:

This will create a four byte character string, given the four byte numeric value *Long*.

Mkl$ is often used when writing long values to sequential files to save on disk space. When the file is later read in, **Cvl** may be used to convert the string back to a long.

See Also:

Cvl

Function: Mkq$

Syntax: **Mkq$** *(Quick)*

Modes: Amiga/Blitz

Description:

This will create a four byte character string, given the four byte numeric value *Quick*.

Mkq$ is often used when writing quick values to sequential files to save on disk space. When the file is later read in, **Cvq** may be used to convert the string back to a quick.

See Also:

Cvq

Function: Cvi

Syntax: **Cvi** *(String$)*

Modes: Amiga/Blitz

Description:

Cvi returns an integer value equivalent to the left 2 characters of *String$*. This is the logical opposite of **Mki$**.

Example:

```
Print Cvi("AA"): ; Prints 16705
```

See Also:

Mki$

Function: Cvl

Syntax: **Cvl** *(String$)*

Modes: Amiga/Blitz

Description:

Cvl returns a long value equivalent to the left 4 characters of *String$*. This is the logical opposite of **Mkl$**.

See Also:

Mkl$

Function: Cvq

Syntax: **Cvq** *(String$)*

Modes: Amiga/Blitz

Description:

Cvq returns a quick value equivalent to the left 4 characters of *String$*. This is the logical opposite of **Mkq$**.

See Also:

Mkq$

Function: Len

Syntax: **Len** *(String$)*

Modes: Amiga/Blitz

Description:

Len returns the length of the string *String$*.

Example:

```
Print Len("Hippo"): ; Will print 5
```

Function: **UnLeft$**

Syntax: **UnLeft$** *(String$,Length)*

Modes: Amiga/Blitz

Description:

UnLeft$() removes the rightmost *Length* characters from the string *String$.*

Example:

```
Print UnLeft$("GoodBye",3): ; Will print Good
MouseWait
```

See Also:

Left$

Function: **UnRight$**

Syntax: **UnRight$** *(String$,Length)*

Modes: Amiga/Blitz

Description:

UnRight$() removes the leftmost *Length* characters from the string *String$.*

Example:

```
Print UnRight$("GoodBye",4): ; Will print Bye
```

Function: **StripLead$**

Syntax: **StripLead$** *(String$,Expression)*

Modes: Amiga/Blitz

Description:

StripLead$ removes all leading occurrences of the ASCII character specified by *Expression* from the string *String$.*

Example:

```
Print StripLead$("AABBAAB",65)     ;Will print BBAAB
```

See Also:

StripTrail$

Function: StripTrail$

Syntax: **StripTrail$** *(String$,Expression)*

Modes: Amiga/Blitz

Description:

StripTrail$ removes all trailing occurrences of the ASCII character specified by *Expression* from the string *String$.*

Example:

```
Print StripTrail$("AABBAAB",66): ;Will print AABBAA
```

See Also:

StripLead$

Function: LSet$

Syntax: **LSet$** *(String$,Characters)*

Modes: Amiga/Blitz

Description:

This function returns a string of *Characters* characters long. The string *String$* will be placed at beginning of this string. If *String$* is shorter than *Characters* the right hand side is padded with spaces. If it is longer, it will be truncated.

Example:

```
Print LSet$("Guy Fawkes",6): ; Will print "Guy Fa"
Print LSet$("Guy",6):         ; Will print "Guy   "
```

See Also:

RSet$**, **Centre$

Function: RSet$

Syntax: **RSet$** *(String$,Characters)*

Modes: Amiga/Blitz

Description:

This function 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.

Example:

Print RSet$("Guy Fawkes",6): *; Will print "Fawkes"*
Print RSet$("Guy",6):   *; Will print "   Guy"*

See Also:

LSet$, Centre$

Function: **Centre$**

Syntax: **Centre$** *(String$,Characters)*

Modes: Amiga/Blitz

Description:

This function returns a string of *Characters* characters long. The string *String$* will be centered in the resulting string. If *String$* 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.

Example:

Print Centre$("Guy Fawkes",6): *; Will print "y Fawk"*
Print Centre$("Guy",6):   *; Will print " Guy "*

See Also:

LSet$, RSet$

Function: **LCase$**

Syntax: **LCase$** *(String$)*

Modes: Amiga/Blitz

Description:

This function returns the string *String$* converted into lowercase.

Example:

Print LCase$("ABCDEFG"): *; Prints abcdefg*

See Also:

UCase$

Function: **UCase$**

Syntax: **UCase$** *(String$)*

Modes: Amiga/Blitz

Description:

This function returns the string *String$* converted to uppercase.

Example:

Print UCase$("hijklm"): ; *Prints HIJKLM*

See Also:

Lcase$

Function: **CaseSense**

Syntax: **CaseSense** *On| Off*

Modes: Amiga/Blitz

Description:

CaseSense allows you to control 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.

See Also:

Instr, Replace$

Function: **Val**

Syntax: **Val** *(String$)*

Modes: Amiga/Blitz

Description:

This functions 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.

Example:

```
Print Val("1234") ; Will Print 1234
Print Val("-23") ; Prints -23
Print Val("One hundred") ; Will Print 0
```

See Also:

Str$, UStr$

Function: **Str$**

Syntax: **Str$** *(Expression)*

Modes: Amiga/Blitz

Description:

This 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.

Example:

```
a$=Str$(12345)
Print Len(a$)   ; Prints 5
```

See Also:

Val, UStr$, Format

Function: **UStr$**

Syntax: **UStr$** *(Expression)*

Modes: Amiga/Blitz

Description:

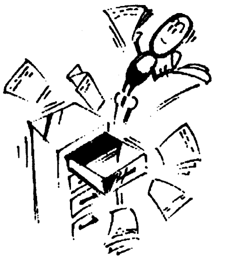
This returns a string equivalent of the numeric value *Expression*. This now allows you to perform string operations on this string.

Unlike **Str$**, **UStr$** is not affected by any active **Format** commands.

See Also:

Val, Str$, Format

7. File Access



Blitz 2 supports 2 modes of file access - sequential, and random access. The following section covers the Blitz 2 commands that open, close and operate on these two types of files.

Function: **OpenFile**

Syntax: OpenFile *(File#,Filename$)*

Modes: Amiga

Description:

OpenFile 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 by **OpenFile**.

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.

Example:

```
;
; random access file program example
;

If OpenFile(0,"ram:test")      ;open random access file.
 MaxLen c$=32                  ;set maximum length of c$
    Fields 0,a,b,c$            ;set up fields in a record
    a=10                       ;initialize some variables...
    b=16
    c$="Hello There!"
    Put 0,0                    ;write record 0
    CloseFile 0                ;close the file
    If OpenFile(0,"ram:test")  ;reopen file
        Fields 0,a,b,c$        ;set up fields again
        a=0                    ;clear variables
        b=0
        c$=""
        Get 0,0                ;read record 0
        NPrint "a=",a," b=",b," c$=",c$
        CloseFile 0            ;close the file
        MouseWait
        End
    EndIf
EndIf

NPrint "Couldn't open ram:test" ;file open failed!
```

```
MouseWait
```

See Also:

CloseFile, Fields, Get, Put, MaxLen

Function: **ReadFile**

Syntax: ReadFile *(File#,Filename$)*

Modes: Amiga

Description:

ReadFile opens an already existing file specified by *Filename$* for sequential reading. If the specified file was successfully opened, **ReadFile** will return true (-1), otherwise **ReadFile** will return false (0).

Once a file is open using **ReadFile**, **FileInput** may be used to read information from it.

Example:

```
;
; read file program example
;

If WriteFile(0,"ram:test")            ;try to write file...
    FileOutput 0                      ;send print statements to file 0
    Print "Hello!"                    ;write "Hello!" to file
    CloseFile 0                       ;close the file
    DefaultOutput                     ;use default output.
    If ReadFile(0,"ram:test")         ;try to read file...
        FileInput 0                   ;get input from file 0
        NPrint Edit$(80)              ;read from file and print it out
        CloseFile 0                   ;close file
        DefaultInput                  ;normal input
        MouseWait
        End
    EndIf
EndIf

NPrint "Couldn't open ram:test!"      ;file open failed!

MouseWait
```

See Also:

CloseFile, WriteFile, FileInput, FileOutput

Function: WriteFile

Syntax: WriteFile *(File#,Filename$)*

Modes: Amiga

Description:

WriteFile 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, **WriteFile** will return false (0).

A file opened using **WriteFile** may be written to by using the **FileOutput** command.

See Also:

CloseFile, ReadFile, FileInput, FileOutput

Statement: CloseFile

Syntax: CloseFile *File#*

Modes: Amiga

Description:

CloseFile is 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.

See Also:

OpenFile, ReadFile, WriteFile

Statement: Fields

Syntax: Fields *File#,Var[,Var...]*

Modes: Amiga/Blitz

Description:

Fields is used to set up fields of a random access file record. Once **Fields** is executed, **Get** and **Put** may be 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 initialized 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.

Example:

```
;
; put and get random access file program example
;

If OpenFile(0,"ram:test")          ;open random access file.
    MaxLen c$=32                   ;set maximum length of c$
    Fields 0,a.f,c$,b.w            ;set up fields in a record
    a=Sqr(Pi)                      ;initialize some variables...
    b=16
    c$="RANDOM ACCESS!"
    Put 0,0                        ;write record 0
    CloseFile 0                    ;close the file
    If OpenFile(0,"ram:test")      ;reopen file
        Fields 0,a,b,c$            ;set up fields again
        a=0                        ;clear variables
        b=0
        c$=""
        Get 0,0                    ;read record 0
        NPrint "a=",a," b=",b," c$=",c$
        CloseFile 0                ;close the file
        MouseWait
        End
    EndIf
EndIf
NPrint "Couldn't open ram:test" ;file open failed!
MouseWait
```

See Also:

OpenFile, CloseFile, Get, Put, MaxLen

Statement: **Put**

Syntax: Put *File#,Record*

Modes: Amiga

Description:

Put is 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.

See Also:

OpenFile, CloseFile, Fields, Get

Statement: **Get**

Syntax: Get *File#,Record*

Modes: Amiga

Description:

Get is used to transfer information from a particular record of a random access file into a variable list set up by the **Fields** command. Only records which also exist may be 'got'.

See Also:

OpenFile, CloseFile, Fields, Put

Statement: **FileOutput**

Syntax: FileOutput *File#*

Modes: Amiga/Blitz

Description:

The **FileOutput** command 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**).

See Also:

WriteFile, CloseFile

Statement: **FileInput**

Syntax: FileInput *File#*

Modes: Amiga/Blitz

Description:

The **FileInput** command 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 appropropraite channel (eg: **DefaultInput** or **WindowInput**).

See Also:

ReadFile, CloseFile

Statement: FileSeek

Syntax: FileSeek *File#,Position*

Modes: Amiga

Description:

FileSeek 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.

Example:

```
;
; file seek random access file program example
;

If WriteFile(0,"ram:test")          ;create new file
     FileOutput 0                     ;send print there...
     NPrint "Hello!"              ;print something!
     CloseFile 0                      ;close file
     If OpenFile(0,"ram:test")        ;open file again
          FileSeek 0,Lof(0)            ;fileseek to end of the file
          NPrint "There!"         ;add to the file
          CloseFile 0                  ;close file again
          DefaultOutput                ;send output back to normal
          If ReadFile(0,"ram:test")     ;open file for reading
               FileInput 0              ;get input from file
               NPrint Edit$(80)           ;read file and print to screen
               NPrint Edit$(80)           ;ditto
          If ReadFile(0,"ram:test")     ;open file for reading
               FileInput 0              ;get input from file
               NPrint Edit$(80)           ;read file and print to screen
               NPrint Edit$(80)           ;ditto
               MouseWait
               End
          EndIf
     EndIf
EndIf

NPrint "Couldn't open ram:test!"          ;file open failed!

MouseWait
```

See Also:

OpenFile, CloseFile, Lof, Eof, Loc

Function: **Lof**

Syntax: Lof *(File#)*

Modes: Amiga

Description:

Lof will return the length, in bytes, of the specified file.

See Also:

OpenFile, CloseFile, Eof, Loc

Function: **Eof**

Syntax: Eof *(File#)*

Modes:Amiga

Description:

The **Eof** function 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 it's length, while any further reading from the file will cause an error.

Example:

```
;
; random access file program example
;

If WriteFile(0,"ram:test")    ;create a new file
   FileOutput 0               ;send print to the file...
   For k=1 To Rnd(50)+50      ;print a random number of
   Print Chr$(Rnd(26)+65)     ;random alphabetic characters
   Next                       ;
   CloseFile 0                ;close the file
   DefaultOutput              ;send output back to screen
   If ReadFile(0,"ram:test")  ;open file for reading
      FileInput 0             ;get input from file
      While NOT Eof(0)        ;while end of file not reached...
         Print Inkey$         ;print next character from file
      Wend                    ;and back for more
      MouseWait
      End
   EndIf
EndIf

NPrint "Unable to open ram:test"   ;couldn't open file

End
```

See Also:

Lof, Loc

Function: **Loc**

Syntax: Loc *(File#)*

Modes: Amiga

Description:

Loc 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.

See Also:

Lof, Eof

Statement: **DosBuffLen**

Syntax: DosBuffLen *Bytes*

Modes: Amiga/Blitz

Description:

All Blitz 2 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 2048 byte buffer. However, if memory is tight this buffer size may be lowered using the **DosBuffLen** command.

Statement: **KillFile**

Syntax: KillFile *Filename$*

Modes: Amiga

Description:

The **KillFile** command will simply attempt to delete the specified file. No error will be returned if the file could not be deleted.

Statement: **CatchDosErrs**

Syntax: CatchDosErrs

Modes: Amiga/Blitz

Description:

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 2 window.

The window you wish dos error requesters to open on should be the currently used window at the time **CatchDosErrs** is executed.

Example:

```
;
; catdoserrs example program
;
Screen 0,3
Window 0,0,12,320,DispHeight-12,$1008,"My Window",1,2
CatchDosErrs                    ;trap dos errs to our window!
If ReadFile(0,"dummydev:dummyfile")    ;nonsense device
 ;
Else
 Print "Can't open file!"
EndIf
Repeat                          ;wait...
Until WaitEvent=$200                ;for window closed.
```

Statement: **ReadMem**

Syntax: ReadMem *File#,Address,Length*

Modes: Amiga

Description:

ReadMem allows you to read a number of bytes, determined by *Length*, into an absoulte 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!

See Also:

WriteMem

Statement: **WriteMem**

Syntax: WriteMem *File#,Address,Length*

Modes: Amiga

Description:

WriteMem 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#*.

See Also:

ReadMem

8. Compiler Directives

The following section refers to the Blitz 2 Compiler Directives, commands which affect how a program is compiled. Conditional compiling, macros, include files and more are covered in this chapter

Directive: **USEPATH**

Syntax: USEPATH *Pathtext*

Modes: Amiga/Blitz

Description:

USEPATH 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 Blitz2 encounters a variable starting with the backslash character ('\'), it simply inserts the current USEPATH text before the backslash.

See Also:

NEWTYPE

Directive: **BLITZ**

Syntax: BLITZ

Modes: Amiga/Blitz

Description:

The **BLITZ** directive is used to enter Blitz mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz 2 Programmers Guide.

See Also:

AMIGA, QAMIGA

Directive: AMIGA

Syntax: AMIGA

Modes: Amiga/Blitz

Description:

The **AMIGA** directive is used to enter Amiga mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz 2 Programmers Guide.

See Also:

BLITZ, QAMIGA

Directive: QAMIGA

Syntax: QAMIGA

Modes: Amiga/Blitz

Description:

The **QAMIGA** directive is used to enter Quick Amiga mode. For a full discussion on Amiga/Blitz mode, please refer to the programming chapter of the Blitz 2 Programmers Guide.

See Also:

BLITZ, AMIGA

Directive: INCLUDE

Syntax: INCLUDE *Filename*

Modes: N/A

Description:

INCLUDE is 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 (ie: saved from the Blitz 2 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.

See Also:

XINCLUDE, INCBIN

Directive: XINCLUDE

Syntax: XINCLUDE *Filename*

Modes: N/A

Description:

XINCLUDE stands for exclusive include. **XINCLUDE** works identically to **INCLUDE** with the exception that **XINCLUDE**'d files are only ever included once. For example, if a program has 2 **XINCLUDE** statements with the same *filename*, only the first **XINCLUDE** will have any effect.

INCDIR may be used to specify a path for *Filename*.

Filename may be optionally quote enclosed to avoid tokenisation problems.

Example:

```
XINCLUDE incfilename$     ;this will do nothing...'incfile' has already been ;included
```

See Also:

INCLUDE, INCBIN

Directive: INCBIN

Syntax: INCBIN *Filename*

Modes: N/A

Description:

INCBIN 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.

INCDIR may be used to specify an AmigaDos path for *Filename*.

Filename may be optionally quote enclosed to avoid tokenisation problems.

Directive: INCDIR

Syntax: INCDIR *Pathname*

Modes: Amiga/Blitz

Description:

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.

Pathname may be optionally quote enclosed to avoid tokenisation problems.

Example:

```
INCDIR ":Myincs/"
INCLUDE mysource.src
```

See Also:

INCLUDE, XINCLUDE, INCBIN

Directive: CNIF

Syntax: CNIF *Constant Comparison Constant*

Modes: N/A

Description:

CNIF allows you to conditionally compile a section of program code based on a comparison of 2 constants. *Comparison* should be one of '<', '>', '=', '<>', '<=' or '>='. 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.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on conditional compiling.

Example:

```
;
; conditional debugging example
;

#debugit=1           ;a debug flag.

For k=1 To 10        ;start of loop
  CNIF #debugit=1      ;is debug flag=1 ?
  NPrint k           ;yes, print out value of 'k'
  CEND               ;end of conditional compiling.
Next

MouseWait
```

See Also:

CEND, CELSE, CSIF

Directive: **CEND**

Syntax: CEND

Modes: N/A

Description:

CEND marks the end of a block of conditionally compiled code. **CEND** must always appear somewhere following a **CNIF** or **CSIF** directive.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on conditional compiling.

See Also:

CNIF, CSIF, CELSE

Directive: **CSIF**

Syntax: CSIF *"String" Comparison "String"*

Modes: N/A

Description:

CSIF allows you to conditionally compile a section of program code based on a comparison of 2 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.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on conditional compiling.

Example:

```
;
; macro example program with cerr
;

Macro test              ;define test macro!
CSIF `1=""              ;check parameter
 CERR "Illegal Macro Parameter"  ;generate error if null!
CEND                    ;NPrint "`1"                ;print parameter
End Macro               ;end of macro definition

!test{hello}            ;this will compile OK
```

```
!test                    ;this will generate an error!
```

See Also:

CEND, CNIF, CELSE

Directive: **CELSE**

Syntax: CELSE

Modes: N/A

Description:

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.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on conditional compiling.

See Also:

CNIF, CSIF, CEND

Directive: **CERR**

Syntax: CERR *Errormessage*

Modes: N/A

Description:

CERR 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.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on conditional compiling.

Directive: **Macro**

Syntax: Macro *Macroname*

Modes: N/A

Description:

Macro is 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.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on macros.

Example:

```
;
; simple macro program example
;

Macro test                ;start of 'test' macro definition
 NPrint "Hello!"          ;macro contents...
 NPrint "This is a Macro!"   ;...
End Macro                 ;end of 'test' macro

!test                 ;insert macro...!test                  ;insert macro

MouseWait
```

See Also:

End Macro

Statement: **End Macro**

Syntax: End Macro

Modes: N/A

Description:

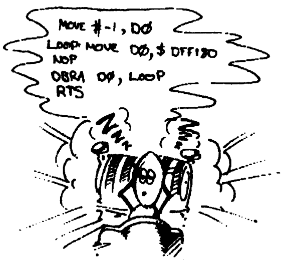
End Macro is used to finish a macro definition. Macro definitions are set up using the **Macro** command.

Please refer to the Programming chapter of the Blitz 2 Programmers Guide for more information on macros.

See Also:

Macro

9. Assembler



This section will cover commands related to Blitz 2's in-line assembler. It is assumed that readers of this section are already knowledgable in 68000 assembly language, as no attempt will be made to teach this subject.

Blitz 2's assembler is very easy to use. All 68000 mnemonics are tokenised as if they were BASIC keywords, and are assembled into machine code when a program is compiled. 68000 code may be intermixed freely with basic, though of course care must be taken not to upset the system.

If you are wanting to use the Blitz 2 assembler for writing straight machine code programs, then you are free to treat Blitz 2 as if it was simply an assembler instead of a compiler. In fact, if you enable runtime error checking, Blitz 2 will even attempt to trap any GURU's in your code!However, if you are wanting to intermix assembly language with BASIC, there are some important rules you must follow:

* Address registers A4-A6 must be preserved and restored by any assembly language routines. Blitz 2 uses A5 as a global variable base, A4 as a local variable base, and tries to keep A6 from having to be re-loaded too often.

The Blitz 2 assembler does have some limitations:

* The Absolute Short addressing mode is not supported.

* Short Branches are not supported.

* Any assembler expressions MUST use curly brackets ('{' and '}') to force operator precedence.

Apart from this, the Blitz 2 assembler operates identically to most commercially available assemblers.

Statement: **DC**

Syntax: **DC** *[.Size] Data[,Data...]*

Description:

DC stands for 'define constant', and may be used to define areas of data for assembler programs.

Statement: **DCB**

Syntax: DCB *[.Size] Repeats,Data*

Description:

DCB stand for 'define constant block'. **DCB** allows you to insert a repeating series of the same value into your assembler programs.

Statement: **DS**

Syntax: Ds *[.Size] Length*

Description:

DS stands for 'define storage'. This simply inserts a 'gap' into a program, which may be used as a data storage area. The constents of DS storage areas will be unpredictable when a program is first run.

Statement: **EVEN**

Syntax: EVEN

Description:

EVEN allows you to word align Blitz 2'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.

Statement: **GetReg**

Syntax: GetReg *68000 Reg,Expression*

Description:

GetReg 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.

GetReg should only be used to transfer expressions to one of the 8 data registers (d0-d7).

GetReg will use the stack to temporarily store any registers used in calculation of the expression.

Statement: **PutReg**

Syntax: PutReg *68000 Reg, Variable*

Description:

PutReg may be 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.

Statement: **SysJsr**

Syntax: SysJsr *Routine*

Description:

SysJsr allows you to call any of Blitz 2's system routines from your own program. *Routine* specifies a routine number to call.

Statement: **TokeJsr**

Syntax: TokeJsr *Token[,Form]*

Description:

TokeJsr allows you to call any of Blitz 2'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. A full list of all token numbers with their various forms will be available shortly from Acid Software.

Statement: **ALibJsr**

Syntax: ALibJsr *Token[,Form]*

Description:

ALibJsr is only used when writing Blitz 2 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 programmers guide for more information on library writing.

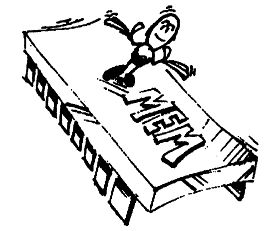
Statement: **BLibJsr**

Syntax: BLibJsr *Token[,Form]*

Description:

BLibJsr is only used when writing Blitz 2 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 programmers guide for more information on library writing.

10. Memory Access



This section deals with low-level commands which allow you access to the Amiga's memory.

Be very careful when using any of the commands in this section, as it is very easy to crash you Amiga by careless **Poke**ing or **Call**ing.

Statement: **Poke**

Syntax: Poke *[.Type] Address,Data*

Modes: Amiga/Blitz

Description:

The **Poke** command will place the specified *Data* into the 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, you would use **Poke.b**; to poke a word into memory you would use **Poke.w**; and to poke a long word into memory you would 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.

See Also:

Peek, Peeks$, Call

Function: **Peek**

Syntax: Peek *[.Type](Address)*

Modes: Amiga/Blitz

Description:

The **Peek** function 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.

See Also:

Poke, Peeks$, Call

Function: **Peeks$**

Syntax: Peeks$ *(Address,length)*

Modes: Amiga/Blitz

Description:

Peeks$ will return a string of characters corresponding to bytes peeked from consective memory locations starting at *Address*, and *Length* characters in length.

See Also:

Peek, Poke, Call

Statement: **Call**

Syntax: Call *Address*

Modes: Amiga/Blitz

Description:

Call will cause program flow to be transferred to the absolute memory location specified by *Address*. PLEASE NOTE! **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 2 program.

Please refer to the 'Assembler' section of the reference guide for the rules machine code programs must follow to operate correctly within the Blitz 2 environment.

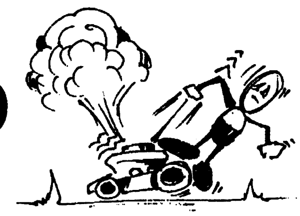
Example:

```
;
; a machine code example
;
a.l=AllocMem_ (14,1)
;read machine code and poke it in:
For k=0 To 12 Step 2
  Read w.w
  Poke.w a+k,w
Next
;call machine code:
Call a
MouseWait
;free up allocated memory:
FreeMem_ a,14
;
;a machine code program...
Data.w $70ff,$33c0,$00df,$f180,$51c8,$fff8,$4e75
```

See Also:

Poke, Peek, Peeks$

11. 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.

Statement: **WBStartup**

Syntax: WBStartup

Modes: Amiga/Blitz

Description:

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.

Function: **NumPars**

Syntax: NumPars

Modes: Amiga/Blitz

Description:

The **NumPars** function 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:

myprog file1 file2

In this case, **NumPars** would return the value '2' - 'file1' and 'file2' beng 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 it's '.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 Arguement' menu item in the 'COMPILER' menu allows you to test out CLI parameters.

Example:

```
;
; numpars program example
;

;before running this program, enter several items of text, space
;separated, into the 'CLI Arguement' requester.

For k=1 To NumPars
 NPrint Par$(k)
Next

MouseWait
```

See Also:

Pars$

Function: **Par$**

Syntax: Par$ *(Parameter)*

Modes: Amiga/Blitz

Description:

Par$ return a string equivalent to a parameter passed to an executable file through either the CLI or Workbench. Please refer to **NumPars** for more information on parameter passing.

Statement: **CloseEd**

Syntax: CloseEd

Modes: Amiga/Blitz

Description:

The **CloseEd** statement will cause the Blitz 2 editor screen to 'close down' when programs are executed from within Blitz 2. This may be useful when writing programs which use a large amount of chip memory, as the editor screen itself occupies about 40K of chip memory.

CloseEd will have no effect on executable files run outside of the Blitz 2 environment.

Example:

```
;
; closeed program example
;

CloseEd

Print "Hello...The editor screen has gone!"
```

MouseWait

See Also:

NoCli

Statement: NoCli

Syntax: NoCli

Modes: Amiga/Blitz

Description:

NoCli will prevent the normal 'Default Cli' from opening when programs are executed from within Blitz 2. **NoCli** has no effect on executable files run outside of the Blitz 2 environment.

See Also:

CloseEd

12. Object Handling

Objects are Blitz 2's way of controlling data concerned with windows, shapes etc. The following section covers the commands available to operate on such objects.

Statement: **Use**

Syntax: Use *Objectname Object#*

Modes: Amiga/Blitz

Description:

Use will cause the Blitz 2 object specified by *Objectname* and *Object#* to become the currently used object.

Example:

```
;
; screens and windows program example
;

Screen 0,3                    ;open a screen & 4 windows...

Window 1,0,0,160,100,$100f,"Window 1",1,2
Window 2,160,0,160,100,$100f,"Window 2",1,2
Window 3,0,100,160,100,$100f,"Window 3",1,2
Window 4,160,100,160,100,$100f,"Window 4",1,2

For k=1 To 4                  ;start of loop
 Use Window k                   ;use window 'k'
 NPrint "Currently using"         ;output text...
 NPrint "Window#:",k
Next                         ;end of loop

MouseWait
```

See Also:

Free

Statement: **Free**

Syntax: Free *Objectname Object#*

Modes: Amiga/Blitz

Description:

Free is used to free a Blitz 2 object. Any memory consumed by the object's existance will be free'd up, and in the case of things such as windows and screens, the display may be altered.
Attempting to free a non-existant object will have no effect.

Example:

```
;
; screens and windows program example
;

Screen 0,3                       ;open intuition screen & 4 windows...

Window 1,0,0,160,100,$f,"Window 1",1,2
Window 2,160,0,160,100,$f,"Window 2",1,2
Window 3,0,100,160,100,$f,"Window 3",1,2
Window 4,160,100,160,100,$f,"Window 4",1,2

c=0                         ;counter for number of windows closed
Repeat                        ;repeat...
 a.l=WaitEvent               ;wait for something to happen
 If a=512                  ;close window ?
  Free Window EventWindow       ;Yes, free window...
  c+1                     ;and increment counter
 EndIf
Until c=4                    ;until all windows closed.
```

See Also:

Use

Function: USED

Syntax: Used ObjectName

Modes: Amiga/Blitz

Description:

Used returns the currently used object number. This is useful for routines which need to operate on the currently used object, also interrupts should restore currently used object settings.

Example:

```
;
; used example
;
BitMap 0,320,200,1
BitMap 1,320,200,1
Use BitMap 0

NPrint Used BitMap ;used returns currently used object number

MouseWait
```

See also:

Use

Function: **Addr**

Syntax: Addr *Objectname(Object#)*

Modes: Amiga/Blitz

Description:

Addr is a low-level function allowing advanced programmers the ability to find where a particular Blitz 2 object resides in RAM. An appendix at the end of this manual lists all Blitz 2 object formats.

Example:

```
;
; object addr program example
;

Screen 0,3

Window 0,0,0,320,200,$100f,"My Window!",1,2

NPrint "Window object 0 resides at:",Addr Window(0)
NPrint "Intuition Window structure is at:",Peek.l(Addr Window(0))

MouseWait
```

Function: **Maximum**

Syntax: Maximum *Objectname*

Modes: Amiga/Blitz

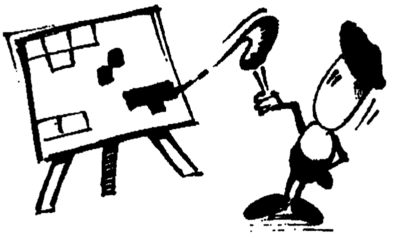
Description:

The **Maximum** function allows a program to determine the 'maximum' setting for a particular Blitz 2 object. Maximum settings are entered into the 'OPTIONS' requester, accessed through the 'COMPILER' menu of the Blitz 2 editor.

Example:

```
;
; maximum program example
;

NPrint "Maximum Windows available:",Maximum Window

MouseWait
```

13. BitMaps



Blitz 2 BitMap objects are used primarily for the purpose of rendering graphics. Most commands in Blitz 2 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.

Statement: **BitMap**

Syntax: **BitMap** *BitMap#,Width,Height,Depth*

Modes: Amiga/Blitz

Description:

BitMap creates and initializes 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.

Example:

```
;
; a bitmap program example
;
BitMap 0,320,200,3  ;A standard lo-res, 8 colour BitMap, Now
                    ;currently used
Circlef 160,100,50,3    ;draw something onto the used BitMap
Screen 0,0,0,320,200,3,0,"My Screen",1,2,0    ;Attach BitMap to Screen
MouseWait
End
```

See Also:

Use BitMap, Free BitMap

Statement: **Use BitMap**

Syntax: **Use BitMap** *BitMap#*

Modes: Amiga/Blitz

Description:

Use 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.

See Also:

BitMap, Free BitMap

Statement: **Free BitMap**

Syntax: **Free BitMap** *BitMap#*

Modes: Amiga/Blitz

Description:

Free 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.

See Also:

BitMap, Use BitMap

Statement: **CopyBitMap**

Syntax: CopyBitMap *BitMap#,BitMap#*

Modes: Amiga/Blitz

Description:

CopyBitMap will make 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.

Statement: **ScreensBitMap**

Syntax: **ScreensBitMap** *Screen#,BitMap#*

Modes: Amiga/Blitz

Description:

Blitz 2 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.

Example:

```
;
; using a screen's bitmap program example
;
Screen 0,3,"My Screen"        ;A Simple Screen.
ScreensBitMap 0,0             ;pick up it's BitMap...
Circlef 160,100,50,3
MouseWait
End
```

See Also:

Screen

Statement: **LoadBitMap**

Syntax: **LoadBitMap** *BitMap#,Filename$[,Palette#]*

Modes: Amiga

Description:

LoadBitMap allows you to load an ILBM IFF graphic into a previously initialized 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.

Example:

```
;
; loadbitmap from disk and display program example
;
Screen 0,3,"My Screen"
ScreensBitMap 0,0
LoadBitMap 0,"MyPic.iff",0
Use Palette 0
MouseWait
End
```

Statement: **SaveBitMap**

Syntax: **SaveBitMap** *BitMap#,Filename$[,Palette#]*

Modes: Amiga

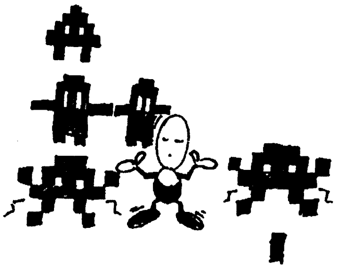
Description:

SaveBitMap allows you to save a bitmap to disk in ILBM IFF format. An optional palette may also be saved with the IFF.

Example:

```
;
; saving a bitmap to disk program example
;
Screen 0,3,"My Screen"
ScreensBitMap 0,0
Circlef 160,100,50,3
SaveBitmap 0,"MyBitMap.iff"          ;create an IFF!
End
```

14. Shapes



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.

Many commands are available for the purpose of drawing shapes onto a bitmap. These commands use the Amiga's blitter chip to achieve this, and are therefore very fast. The process of putting a shape onto a bitmap using the blitter is often referred to as 'blitting' a shape. The speed at which a shape is blitted is important when you are writing animations routines, as the smoothness of any animation will be directly affected by how long it takes to draw the shapes involved in the animation.

There are 2 main factors which affect the speed at which a shape is blitted - it's size, and the technique used to actually blit the shape. Let's have a look at how the size of a shape affects it's 'blit speed'.

Obviously, larger shapes take longer to blit than smaller shapes. Not so obviously, shapes with more colours in them take longer to blit than shapes with fewer colours. A 2 bitplane (4 colour) shape will take twice as long to blit as a 1 bitplane (2 colour) shape. A 3 bitplane (8 colour) shape will take three times as long to blit as a 1 bitplane shape and so on.

The technique used to blit a shape also affects it's speed. The fastest blitting command you can use is the simple **Blit** command. However, this provides no way of erasing of shapes to allow for movement. **QBlit** is the fastest way to achieve this. **BBlit** is the slowest of the blit commands, but also the most versatile and least memory intensive.

One of a programmers most difficult tasks is that of achieving acceptable compromises. This is especially true in the case of blitting shapes. While it certainly would be nice to have 50 individual 64 colour shapes flying smoothly around the screen, the Amiga is not really up to it. Therefore, the programmer must decide on an acceptable compromise - Should less shapes be used? Maybe less colours? A combination of both? The answer will depend on what you as a programmer decide is best in the situation.

Statement: **LoadShape**

Syntax: LoadShape *Shape#,Filename$[,Palette#]*

Modes: Amiga

Description:

LoadShape 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.

Example:

```
;
; simple blit shape example
;
```

```
Screen 0,3                 ;open an intuition screen
ScreensBitMap 0,0          ;get its bitmap
LoadShape 0,"MyShape.iff",0  ;load a shape from disk
Use Palette 0              ;use its palette
Blit 0,0,0                 ;blit it onto the screen
MouseWait
```

See Also:

LoadShapes, SaveShape, SaveShapes

Statement: **SaveShape**

Syntax: SaveShape *Shape#,Filename$,Palette#*

Modes: Amiga

Description:

SaveShape will create 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.

See Also:

SaveShapes, LoadShape, LoadShapes

Statement: **LoadShapes**

Syntax: LoadShapes *Shape#[,Shape#],Filename$*

Modes: Amiga

Description:

LoadShapes 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 the **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.

See Also:

SaveShapes, LoadShape, LoadShapes

Statement: **SaveShapes**

Syntax: SaveShapes *Shape#,Shape#,Filename$*

Modes: Amiga

Description:

SaveShapes 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.

See Also:

LoadShapes, LoadShape, SaveShape

Statement: **GetaShape**

Syntax: GetaShape *Shape#,X,Y,Width,Height*

Modes: Blitz/Amiga

Description:

GetaShape lets you transfer a rectangular area of the currently used bitmap into the specified shape object. *X, Y, Width* and *Height* specify the area of the bitmap to be picked up and used as a shape.

Example:

```
;
; getashape and randomly blit it example
;
Screen 0,3                    ;an intuition screen
ScreensBitMap 0,0                 ;pick up it's bitmap
Cls                         ;clear bitmap
Boxf 10,10,29,29,2               ;draw some stuff for a shape
Box 12,12,27,27,3
Circlef 20,20,5,4
GetaShape 0,10,10,20,20             ;pick shape 0 up off bitmap
Cls                         ;clear bitmap again

For k=1 To 100                ;start of loop
 Blit 0,Rnd(160)+80,Rnd(100)+50    ;blit shape 0 at random position
Next                        ;end of loop

MouseWait
```

Statement: **CopyShape**

Syntax: CopyShape *Shape#,Shape#*

Modes: Amiga/Blitz

Description:

CopyShape will produce 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.

Statement: **AutoCookie**

Syntax: AutoCookie *On| Off*

Modes: Amiga/Blitz

Description:

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 do, however, consume some of your valuable Chip memory.

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**. This is a good idea if you are not going to be using shapes for blitting - for example, shapes used for gadgets or menus.

See Also:

MakeCookie

Statement: **MakeCookie**

Syntax: MakeCookie *Shape#*

Modes: Amiga/Blitz

Description:

MakeCookie 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.

See Also:

AutoCookie

Function: **ShapeWidth**

Syntax: ShapeWidth *(Shape#)*

Modes: Amiga/Blitz

Description:

The **ShapeWidth** function returns the width, in pixels, of a previously created shape object.

See Also:

ShapeHeight

Function: **ShapeHeight**

Syntax: ShapeHeight *(Shape#)*

Modes: Amiga/Blitz

Description:

The **ShapeHeight** function returns the height, in pixels, of a previously created shape object.

See Also:

ShapeWidth

Statement: **Handle**

Syntax: Handle *Shape#,X,Y*

Modes: Amiga/Blitz

Description:

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 'acrosswards' offset for a handle, the *Y* parameter specifies a 'downwards' offset.

Let's have a look at an example of how a handle works. Assume you have set a shapes *X* handle to 5, and it's *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 actually 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, it's handle is automatically set to 0,0 - it's upper left corner.

See Also:

MidHandle

Statement: MidHandle

Syntax: MidHandle *Shape#*

Modes: Amiga/Blitz

Description:

MidHandle will cause the handle of the specified shape to be set to it's 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.

See Also:

Handle

Statement: XFlip

Syntax: XFlip *Shape#*

Modes: Amiga/Blitz

Description:

The **XFlip** command is one of Blitz 2's powerful shape manipulation commands. **XFlip** will horizontally 'mirror' a shape object, causing the object to be 'turned back to front'.

Example:

```
;
; xflip example
;
Screen 0,3                  ;an intuition screen
ScreensBitMap 0,0           ;it's bitmap
Cls                         ;clear it
Circlef 32,32,32,3          ;draw...
Boxf 32,0,63,63,2           ;some weird shape
GetaShape 0,0,0,64,64       ;pick it up off bitmap
Cls                         ;clear bitmap again
CopyShape 0,1               ;make a copy of shape
XFlip 1                     ;x flip copy
Blit 0,0,0                  ;show original
Blit 1,0,100                ;show flipped copy
MouseWait
```

See Also:

YFlip

Statement: **YFlip**

Syntax: YFlip *Shape#*

Modes: Amiga/Blitz

Description:

The **YFlip** command may be used to vertically 'mirror' a shape object. The resultant shape will appear to have been 'turned upside down'.

Example:

```
;
; yflip example
;
Screen 0,3              ;open an intuition screen
ScreensBitMap 0,0       ;borrow it's bitmap
Cls                     ;clear the bitmap
Circlef 32,32,32,3      ;draw some...
Boxf 0,32,63,63,2       ;weird shape
GetaShape 0,0,0,64,64   ;pick shape 0 up from bitmap
Cls                     ;clear bitmap
CopyShape 0,1           ;make copy of shape
YFlip 1                 ;Y Flip the copy
Blit 0,0,0              ;show original
Blit 1,160,0            ;show flipped copy
MouseWait
```

See Also:

XFlip

Statement: **Scale**

Syntax: Scale *Shape#,X Ratio,Y Ratio[,Palette#]*

Modes: Amiga/Blitz

Description:

Scale is a very powerful command which may be used to 'stretch' or 'shrink' shape objects. The *Ratio* 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.

Example:

```
;
; scale shape example
;
Screen 0,3                 ;An intuition screen
ScreensBitMap 0,0          ;the screens bitmap
Cls                        ;clear the bitmap
For k=7 To 1 Step -1       ;a loop to generate some
Circlef 32,32,k*4,k        ;kind of shape
Next
GetaShape 0,0,0,64,64      ;pick up the shape

For k=1 To 6               ;start of loop
  CopyShape 0,k            ;copy shape
  Scale k,k/4,k/4          ;resize it
Next                       ;end of loop

Cls

For k=1 To 6               ;start of loop
  Blit k,k*32,0            ;show shapes we just generated
Next                       ;end of loop

MouseWait
```

See also:

Rotate

Statement: **Rotate**

Syntax: Rotate *Shape#,Angle Ratio*

Modes: Amiga/Blitz

Description:

The **Rotate** command allows you to rotate a shape object. *Angle Ratio* specifies how much clockwise rotation to apply, and should be in the range zero to one. For instance, an *Angle Ratio* of .5 will cause a shape to be rotated 180 degrees, while an *Angle Ratio* of .25 will cause a shape to be rotated 90 degrees clockwise.

Example:

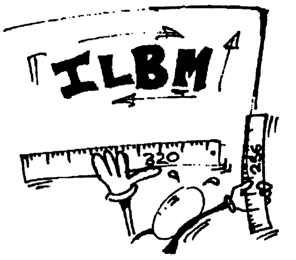
```
;
; rotate shape example with qblit for smooth spinning
;
Screen 0,1
ScreensBitMap 0,0          ;grab it's bitmap
BitMapOutput 0             ;use bitmap for 'Print' commands
Queue 0,1                  ;set up a Queue for the QBlit...
Cls                        ;clear the bitmap
Boxf 0,0,15,63,1           ;draw a rectangle
GetaShape 0,0,0,16,64      ;grab it as a shape
```

```
Cls                   ;clear bitmap
Print "Please Wait"

For k=1 To 64          ;start of loop
 CopyShape 0,k           ;make 64 copies of original shape!
 Rotate k,k/64          ;rotate each copy a little more than last
 MidHandle k             ;and handle in the middle
 Print "."           ;tell user we're doin the job
Next                  ;end of copy loop

Cls               ;clear bitmap

While Joyb(0)=0         ;while joystick button not down...
 For k=1 To 64        ;show all shapes
 VWait            ;wait for top of frame
 UnQueue 0           ;clear the Queue
 QBlit 0,k,160,100      ;Draw next shape
 Next
Wend
```

See Also:

Scale

15. ILBM



ILBM stands for InterLeaved BitMap. This refers to a format many art packages use to store image files in. Electronic Art's excellent DPaint, for example, uses the ILBM format to save it's picture and brush files.

Blitz 2 supplies various commands to examine the attributes of ILBM files.

Statement: **ILBMInfo**

Syntax: ILBMInfo *Filename$*

Modes: Amiga

Description:

ILBMInfo is used to examine an ILBM file. Once ILBMInfo has been executed, **ILBMWidth**, **ILBMHeight** and **ILBMDepth** may be used to examine properties of the image contained in the file.

Function: **ILBMWidth**

Syntax: ILBMWidth

Modes: Amiga

Description:

ILBMWidth will return the width, in pixels, of an ILBM image examined with **ILBMInfo**.

Function: **ILBMHeight**

Syntax: ILBMHeight

Modes: Amiga

Description:

ILBMHeight will return the height, in pixels, of an ILBM image examined with **ILBMInfo**.

Statement: **ILBMDepth**

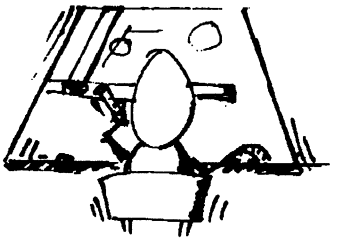
Syntax: ILBMDepth

Modes: Amiga

Description:

ILBMDepth will return the depth, in bitplanes, of an ILBM image examined with **ILBMInfo**.

16. 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.

Statement: **Cls**

Syntax: Cls *[Colour]*

Modes: Amiga/Blitz

Description:

Cls 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'.

Example:

```
;
; simple cls example
;
Screen 0,3              ;open an intuition screen
ScreensBitMap 0,0       ;use it's bitmap
Cls 2                   ;fill bitmap with colour 2
MouseWait
```

Statement: **Plot**

Syntax: Plot *X,Y,Colour*

Modes: Amiga/Blitz

Description:

Plot is used to alter the colour of an individual pixel on the currently used bitmap. *X* and *Y* specify the location of the pixel to be altered, and *Colour* specifies the colour to change the pixel to.

A *Colour* parameter of -1 will cause the pixel at the specified pixel position to be 'inverted'.

Example:

```
;
; simple plot example
;
Screen 0,3              ;an intuition screen
```

```
ScreensBitMap 0,0          ;the screen's bitmap

For x=0 To 319              ;start of loop
 Plot x,100,3              ;what a boring plot!
Next                       ;end of loop

MouseWait
```

See Also:

Point

Function: **Point**

Syntax: Point *(X,Y)*

Modes: Amiga/Blitz

Description:

The **Point** function will 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.

Example:

```
;
; point example
;
Screen 0,3,"HELLO THERE"         ;intuition screen...
ScreensBitMap 0,0               ;and bitmap of screen

For y=0 To 9                    ;one loop...
 For x=0 To 47                  ;another
  Plot x,y,7-Point(x,y)          ;calc inverse colour for plot
 Next                          ;end of x loop
Next                           ;end of y loop

MouseWait
```

See Also:

Plot

Statement: **Line**

Syntax: Line *[X1,Y1,]X2,Y2,Colour*

Modes: Amiga/Blitz

Description:

The **Line** command 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, 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.

Example:

```
;
; line example
;
Screen 0,3                          ;an intuition screen
ScreensBitMap 0,0                   ;it's bitmap

For k=1 To 100                      ;start of loop...
 Line Rnd(320),Rnd(200),Rnd(7)+1    ;random lines!
Next                                ;end of loop
```

Statement: **Box**

Syntax: Box *X1,Y1,X2,Y2,Colour*

Modes: Amiga/Blitz

Description:

The **Box** command draw a rectangular outline onto the currently used bitmap. *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.

Example:

```
;
; simple box example
;
Screen 0,3                          ;intuition screen
ScreensBitMap 0,0                   ;it's bitmap

For k=1 To 100                      ;start of loop
 Box Rnd(320),Rnd(200),Rnd(320),Rnd(200),Rnd(7)+1  ;random boxes
Next                                ;end of loop

MouseWait
```

See Also:

Boxf

Statement: **Boxf**

Syntax: Boxf *X1,Y1,X2,Y2,Colour*

Modes: Amiga/Blitz

Description:

Boxf draws a solid rectangular shape on the currently used bitmap. 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'.

Example:

```
;
; boxf example
;
Screen 0,3,"Hello There"       ;an intuition screen
ScreensBitMap 0,0              ;bitmap of the screen
Boxf 0,0,47,9,-1               ;an inversed box
MouseWait
```

See Also:

Box

Statement: **Circle**

Syntax: Circle *X,Y,Radius[,Y Radius],Colour*

Modes: Amiga/Blitz

Description:

Circle will draw an open circle onto the currently used bitmap. *X* and *Y* specify the mid point of the circle. The *Radius* parameter specifies the radius of the circle. If a *Y Radius* parameter is supplied, then an ellipse may be drawn.

A *Colour* parameter of -1 will cause an 'inverted' circle to be drawn.

Example:

```
;
; circle example
;
Screen 0,3                     ;an intuition screen
ScreensBitMap 0,0              ;bitmap of screen

For k=1 To 10                  ;start of loop
 Circle Rnd(320),Rnd(200),Rnd(100),Rnd(7)+1  ;random circles
Next                           ;end of loop

MouseWait
```

Statement: **Circlef**

Syntax: Circlef *X,Y,Radius[,Y Radius],Colour*

Modes: Amiga/Blitz

Description:

Circlef will draw a filled circle onto the currently used bitmap. *X* and *Y* specify the mid point of the circle - *Colour*, the colour in which to draw the circle. The *Radius* parameter specifies the radius of the circle. If a *Y Radius* parameter is supplied, then an ellipse may be drawn.

A *Colour* parameter of -1 will cause an 'inverted' circle to be drawn.

Example:

```
;
; circlef example
;
Screen 0,3                    ;an intuition screen
ScreensBitMap 0,0

For k=1 To 10                    ;start of loop
 Circlef Rnd(320),Rnd(200),Rnd(100),Rnd(7)+1   ;random circles
Next                             ;end of loop

MouseWait
```

Statement: **Scroll**

Syntax: Scroll *X1,Y1,Width,Height,X2,Y2[,Source BitMap]*

Modes: Amiga/Blitz

Description:

Scroll allows rectangular areas within a bitmap to be moved around. *X1*, *Y1*, *Width* and *Height* specify the position and size of the rectangle to be moved. *X2* and *Y2* specify the position the rectangle is to be moved to.

An optional *Source BitMap* parameter allows you to move rectangular areas from one bitmap to another.

Example:

```
;
; scroll example
;
Screen 0,3,"YEEEEEHHHHHAAAAAA!"    ;an intuition screen
ScreensBitMap 0,0              ;it's bitmap

For k=16 To 192 Step 16          ;start of loop
 Scroll 0,0,320,10,0,k         ;move title bar!
Next                        ;end of loop
MouseWait
```

Statement: **FloodFill**

Syntax: FloodFill *X,Y,Colour [,Border Colour]*

Modes: Amiga/Blitz

Description:

FloodFill 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.

Statement: **FreeFill**

Syntax: FreeFill

Modes: Amiga/Blitz

Description:

FreeFill will deallocate the memory that Blitz 2 uses to execute the commands **Circlef, FloodFill, ReMap** and **Boxf**.

Blitz 2 uses a single monochrome bitmap the size of the bitmap being drawn to to do it's filled routines, by using the FreeFill command this BitMap can be 'freed' up if no more filled commands are to be executed.

17. Palettes, Fades and Cycling

Palette objects are temporary storage areas of RGB and colour cycling information. This information is normally taken from an ILBM IFF file.

Blitz 2 supports colour cycling.

Blitz 2 also supports the ability to 'fade in' or 'fade out' colour palettes in Blitz mode.

Statement: **LoadPalette**

Syntax: LoadPalette *Palette#,Filename$[,Palette Offset]*

Modes: Amiga

Description:

LoadPalette creates and initializes 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 *Palette Offset* 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.

Example:

```
;
; palette program example
;
Screen 0,3                    ;open a simple, 8 colour screen
LoadScreen 0,"picture.iff"    ;load a picture into the screen
LoadPalette 0,"picture.iff"   ;load pictures colours
Use Palette 0                 ;display the colours.
MouseWait
End
```

Statement: **Use Palette**

Syntax: **Use Palette** *Palette#*

Modes: Amiga/Blitz

Description:

Use 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.

Example:

```
;
; loadscreen program example with palette
;
Screen 0,3                 ;open a simple, 8 colour screen
LoadScreen 0,"picture.iff",0   ;load a picture into the screen, and palette as well
Use Palette 0              ;display the colours.
MouseWait
End
```

Statement: **Free Palette**

Syntax: **Free Palette** *Palette#*

Modes: Amiga/Blitz

Description:

Free Palette erases all information in a palette object. That Palette object may no longer be **Use**d or **Cycle**d.

See Also:

Use Palette, LoadPalette

Statement: **PalRGB**

Syntax: PalRGB *Palette#,Colour Register,Red,Green,Blue*

Modes: Amiga/Blitz

Description:

PalRGB 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 **Use Palette** is executed.

Example:

```
;
; setting up a palette program example
;
PalRGB 0,0,6,6,6
PalRGB 0,1,15,15,15
PalRGB 0,2,0,0,0
```

```
PalRGB 0,3,15,15,0
Screen 0,3,"A Manually created palette object!"
Use Palette 0
MouseWait
```

See Also:

Use Palette, RGB, LoadPalette

Statement: **SetCycle**

Syntax: SetCycle *Palette#,Cycle,Low Colour,High Colour [,Speed]*

Modes: Amiga

Description:

SetCycle is used to configure colour cycling information for the **Cycle** command. The *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.

Statement: **Cycle**

Syntax: Cycle *Palette#*

Modes: Amiga

Description:

Cycle will cause 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.

Example:

```
;
; loading a palette and cycling colours program example
;
Screen 0,3                 ;open a simple, 8 colour screen
LoadScreen 0,"picture.iff"     ;load a picture into the screen
LoadPalette 0,"picture.iff"    ;load pictures colours
Use Palette 0              ;display the colours.
Cycle 0
MouseWait
End
```

See Also:

LoadPalette, SetCycle, StopCycle

Statement: **StopCycle**

Syntax: StopCycle

Modes: Amiga

Description:

StopCycle will halt all colour cycling started with the **Cycle** command.

Statement: **Rgb**

Syntax: **Rgb** *Colour Register,Red,Green,Blue*

Modes: Amiga/Blitz

Description:

Rgb 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.

Example:

```
;
; setting a palette colour program example
;
Screen 0,3          ;open up an Intuition Screen
RGB 0,15,0,0         ;this will set background colour to red
MouseWait
```

See Also:

PalRGB, Red, Green, Blue

Function: **Red**

Syntax: **Red** *(Colour Register)*

Modes: Amiga/Blitz

Description:

Red 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.

Red will always return a value in the range zero to fifteen.

Example:

```
;
; red() function program example
;
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
RGB 0,8,4,2
NPrint "Red of colour 0 = ";Red(0)
MouseWait
End
```

See Also:

Green, Blue, RGB

Function: **Green**

Syntax: **Green** *(Colour Register)*

Modes: Amiga/Blitz

Description:

Green 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.

Green will always return a value in the range zero to fifteen.

Example:

```
;
; green() program example
;
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
RGB 0,8,4,2
NPrint "Green of colour 0 = ";Green(0)
MouseWait
End
```

See Also:

Red, Blue, RGB

Function: **Blue**

Syntax: **Blue** *(Colour Register)*

Modes: Amiga/Blitz

Description:

Blue 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.

Blue will always return a value in the range zero to fifteen.

Example:

```
;
; blue() program example
;
Screen 0,3
ScreensBitMap 0,0
BitMapOutput 0
RGB 0,8,4,2
NPrint "Blue of colour 0 = ";Blue(0)
MouseWait
```

See Also:

Red, Green, RGB

Statement: **FadeIn**

Syntax: FadeIn *Palette#[,Rate[,Low Colour, High Colour]]*

Modes: Blitz

Description:

Fadein will cause the colour palette of the currently used slice to be 'faded in' from black up to the RGB values contained in the specified *Palette#*.

Rate# allows you to control the speed of the fade, with 0 being the fastest fade.

Low Colour and *High Colour* allow you to control which colour palette registers are affected by the fade.

Example:

```
;
; fadein example
;
For k=1 To 15          ;set up our own palette object...
  PalRGB 0,k,k,0,15-k
Next

BitMap 0,320,200,4     ;set up a 16 colour bitmap

For k=1 To 100         ;draw 100 random circles
```

```
 Circlef Rnd(320),Rnd(200),Rnd(40),Rnd(15)+1
Next

BLITZ                       ;go into blitz mode
Slice 0,44,320,200,$fff8,4,8,32,320,320 ;a simple slice

For k=0 To 15          ;set all RGBs in slice to black
 RGB k,0,0,0
Next

Show 0        ;show bitmap
VWait 50      ;pause for effect
FadeIn 0,1    ;fade in palette# 0 at a rate of 1
MouseWait
```

See Also:

FadeOut

Statement: **FadeOut**

Syntax: FadeOut *Palette#[,Rate[,Low Colour, High Colour]]*

Modes: Blitz

Description:

Fadeout will cause the colour palette of the currently used slice to be 'faded out' from the RGB values contained in the specified *Palette#* down to black.

Rate# allows you to control the speed of the fade, with 0 being the fastest fade.

Low Colour and *High Colour* allow you 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**.

See Also:

Fadein

Statement: **ASyncFade**

Syntax: ASyncFade *On| Off*

Modes: Amiga/Blitz

Description:

ASyncFade 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 thewhole 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.

See Also:

DoFade, FadeStatus

Statement: **DoFade**

Syntax: DoFade

Modes: Amiga

Description:

DoFade will cause 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 any steps of fading left to perform.

See Also:

ASyncFade, FadeStatus

Function: **FadeStatus**

Syntax: FadeStatus

Modes: Blitz

Description:

FadeStatus is 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 **DoFade**s are required), then **FadeStatus** will return true (-1). If not, **FadeStatus** will return false (0). Please refer to **ASyncFade** and **DoFade** for more information.

See Also:

ASyncFade, FadeIn, FadeOut, DoFade

18. Sound

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 later played back.

Statement: **LoadSound**

Syntax: **LoadSound** *Sound#,Filename$*

Modes: Amiga

Description:

LoadSound 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.

Example:

```
;
; a sound program example
;
LoadSound 0,"Zap.iff"
Sound 0,1
MouseWait
End
```

See Also:

Sound

Statement: **Sound**

Syntax: **Sound** *Sound#,Channelmask[,Vol1[,Vol2...]]*

Description:

Sound 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 one through fifteen.

The following is a list of *Channelmask* values and their effect:

ChannelMask	Channel 0	Channel 1	Channel 2	Channel 3
1	on	off	off	off
2	off	on	off	off
3	on	on	off	off
4	off	off	on	off
5	on	off	on	off
6	off	on	on	off
7	on	on	on	off
8	off	off	off	on
9	on	off	off	on
10	off	on	off	on
11	on	on	off	on
12	off	off	on	on
13	on	off	on	on
14	off	on	on	on
15	on	on	on	on

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.

Example:

```
;
; a very sound program example
;
LoadSound 0,"Mysound.iff"
Sound 0,15,8,16,32,64
MouseWait
End
```

See Also:

LoadSound

Statement: **LoopSound**

Syntax: LoopSound *Sound#,Channelmask[,Vol1[,Vol2...]]*

Modes: Amiga/Blitz

Description:

LoopSound 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**.

Example:

```
;
; loop sound program example
;
LoadSound 0,"MySound.off"        ;load sound and loop info.
LoopSound 0,15
MouseWait
```

Statement: **Volume**

Syntax: **Volume** *Channelmask,Vol1[,Vol2...]*

Modes: Amiga/Blitz

Description:

Volume 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.

Example:

```
;
; sound fader program example
;
LoadSound 0,"MySound.iff"
Sound 0,1

For v=64 To 0 Step -16
  VWait              ;wait a frame
  Volume 1,v         ;set new volume
Next

MouseWait
End
```

See Also:

Sound

Statement: **InitSound**

Syntax: **InitSound** *Sound#,Length[,Period[,Repeat]]*

Modes: Amiga/Blitz

Description:

InitSound initializes 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.

Example:

```
;
; custom waveform program example
;
InitSound 0,32
co.f=Pi/32/2            ;to convert from radians to a '32 degree'
                        ;system.

For k=0 To 31
 SoundData 0,k,Sin(k*co)*127
Next

LoopSound 0,15
MouseWait
```

See Also:

SoundData, Sound

Statement: **SoundData**

Syntax: SoundData *Sound#,Offset,Data*

Description:

SoundData 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.

Example:

```
;
; make a square wave program example
;
InitSound 0,32          ;Get a sound object ready.
```

```
For k=0 To 31          ;Here, we will make a 'Square' waveform.
 If k<16
  SoundData 0,k,127
 Else
  SoundData 0,k,-128
 EndIf
Next

LoopSound 0,15         ;Play the sound.

MouseWait
```

See Also:

InitSound, Sound, LoopSound

Function: **PeekSound**

Syntax: PeekData *(Sound#,Offset)*

Modes: Amiga/Blitz

Description:

PeekSound returns the byte of a sample at the specified offset of the sound object specified.

See Also:

SoundData, InitSound

Statement: **DiskPlay**

Syntax: DiskPlay *Filename$,Channelmask[,Vol1[,Vol2...]]*

Modes: Amiga

Description:

DiskPlay will play 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. The **DiskPlay** 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. Also, **DiskPlay** allows you to play samples of any length, whereas **LoadSound** only allows samples up to 128K in length to be loaded.

For information on the *Channelmask* and *Vol* parameters, please refer to the **Sound** command

Example:

```
;
; diskplay program example
;
DiskPlay "Introduction.iff",1,64
```

See Also:

DiskBuffer, Sound

Statement: **DiskBuffer**

Syntax: **DiskBuffer** *Bufferlen*

Modes: Amiga/Blitz

Description:

DiskBuffer 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.

See Also:

DiskPlay

Statement: **Filter**

Syntax: **Filter** ***On|Off***

Modes: Amiga/Blitz

Description:

Filter may be used to turn on or off the Amiga's low pass audio filter.

Example:

```
;
; filter on program example
;
Filter On
DiskPlay "MySound",1
```

Music Modules

The Soundtracker and Noisetracker format for creating sequenced music has become pretty much an Amiga standard. Blitz 2 supports commands for the loading and playing of songs ('modules') created using Soundtracker or Noisetracker compatible sequencer programs.

Blitz 2 uses module objects to keep track of different pieces of music, allowing you to have more than one module loaded at a time.

Statement: **LoadModule**

Syntax: LoadModule *Module#,Filename$*

Modes: Amiga

Description:

LoadModule loads in from disk a soundtracker/noisetracker music module. This module may be later played back using **PlayModule**.

See Also:

PlayModule, **StopModule**

Statement: **Free Module**

Syntax: **Free Module** *Module#*

Modes: Amiga/Blitz

Description:

Free Module may be used to delete a module object. Any memory occupied by the module will also be free'd.

See Also:

LoadModule

Statement: **PlayModule**

Syntax: PlayModule *Module#*

Modes: Amiga/Blitz

Description:

PlayModule will cause a previously loaded soundtracker/noisetracker song module to be played back.

See Also:

LoadModule, StopModule

Statement: StopModule

Syntax: StopModule

Modes: Amiga/Blitz

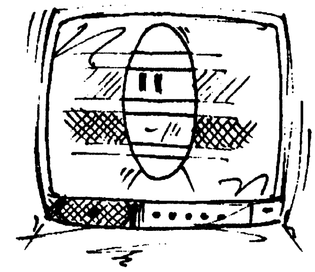
Description:

StopModule will cause any soundtracker/noisetracker modules which may be currently playing to stop.

See Also:

LoadModule, PlayModule

19. Slices



Slices are Blitz 2 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!

Blitz mode's main feature is it's flexible control over the Amiga's display. This control is achieved through the use of slices. 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.

There are some limits placed upon how multiple slices may be arranged:

* Slices must not overlap in any way

* Slices must not be positioned horizontally 'beside' each other. This means multiple slices must be positioned vertically 'on top of' each other.

* When you specify an area for a slice, you only have control over the slices vertical position, it's width and it's height. A slice's horizontal starting position will be automatically calculated in a way which causes the slice to be horizontally centred based on it's width.

* Slices normally require a gap of at least two horizontal lines between the bottom of one slice and the top of another, although there are some situations where this is not necessary.

Here is a simple example of setting up a basic slice driven Blitz mode display:

```
;
BLITZ           ;go into Blitz mode!
Slice 0,44,3     ;set up slice 0
MouseWait
;
```

We wont go too deeply into how the slice command actually works just now, but this example will set up a lo-res eight colour slice - 320 pixels across by either 200 or 256 pixels down, depending on whether you are using an NTSC or PAL machine.

If you type in and compile this example, you will notice that the display contains some fairly random graphics. This is because slices simply control *how* the display appears - they don't control *what* is actually to appear. To use slices to display graphics, a bitmap and some form of the **Show** command is required:

```
;
; slice showing a bitmap program example
;
BLITZ                        ;Go into Blitz mode!
BitMap 0,320,DispHeight,3    ;set up an 8 colour bitmap
Cls 2                        ;fill bitmap with colour 2
Slice 0,44,3                 ;set up a slice
Show 0                       ;show bitmap 0 in the slice
MouseWait
```

Once the bitmap is initialized and **Shown** in this way, any bitmap related commands may be used to render graphics. Here is an example of the **Circlef** command at work in a slice:

```
;
; a functional slice program example
;
BLITZ                          ;go into Blitz mode
BitMap 0,320,DispHeight,3      ;set up an 8 colour bitmap
Slice 0,44,3                   ;set up a slice
Show 0

For k=1 To 100                 ;draw 100 circles.
  Circlef Rnd(320),Rnd(DispHeight),Rnd(10)+5,Rnd(7)+1
Next

MouseWait
```

These examples are all very simple in nature, but illustrate the minimum necessary steps involved in putting single slices to work.

The form of the **Slice** command used in the above examples is a 'quick' form of the standard **Slice** command. Using **Slice** this way limits you to having just one slice active on the display at once. If you are wanting multiple slices, you must use the more complex **Slice** command.

Here's a quick example of multiple slices:

```
;
; multi slice program example
;
BLITZ                          ;Blitz mode! Yeah!
BitMap 0,320,100,3             ;make a bitmap
Cls 2                          ;fill it with colour 2
BitMapOutput 0                 ;we're going to print to it...
Print "Hello - Slice Magic!"   ;like so...
Slice 0,44,160,100,$fff8,3,8,32,320,320 ;whew!
RGB 1,15,15,15                 ;this affects slice 0's palette
RGB 2,8,0,15                   ;so does this
Show 0                         ;show the bitmap
Slice 1,146,320,100,$fff9,3,8,32,320,320   ;more whew!
RGB 1,15,15,0                  ;this affects slice 1's palette
RGB 2,0,8,15                   ;ditto
Show 0                         ;show the bitmap (same one!)
MouseWait
```

Note that the text 'Hello - Slice Magic!' appears at two different places on the display, even though it was only printed once! This is because both slices are **Show**ing the same bitmap, and it is on this bitmap that the text is rendered.

Also note that the top slice is in lo-res mode, whereas the bottom slice is in hi-res mode.

Finally, note that the positioning of the slices follows the rules outlined above. The slices are stacked vertically, and there is a two line gap between them.

One final important word about slices - slice objects can not be individually **Free**'d. This means once a slice is created - it's there for good. The only way to free up slices is to free the lot of them at once using the **FreeSlices** command.

Statement: **Slice**

Syntax:Slice *Slice#,Y,Flags*

Slice *Slice#,Y,Width,Height,Flags,BitPlanes,Sprites,Colours,Width1,Width2*

Modes: Amiga/Blitz

Description:

The **Slice** command is used to create a Blitz 2 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 'downwards' 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 eight 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.

Width1 and *Width2* 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, *Width1* refers to the width of the 'foreground' bitmap, and *Width2* refers to the width of the 'background' bitmap. If a slice is NOT set up to be a dual-playfield slice, both *Width1* and *Width2* 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* setting**	**EffectMax BitPlanes**
$fff8	A standard lo-res slice6
$fff9	A standard hi-res slice4
$fffa	A lo-res, dual-playfield slice6
$fffb	A hi-res, dual-playfiled slice4
$fffc	A HAM slice6 only

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 numbers 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 allowins 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.

Example:

```
;
; slice with circle program example
;
BLITZ                         ;Goodbye OS!
BitMap 0,320,200,3                  ;make a bitmap
Circlef 160,100,50,2                ;draw a circle
Slice 0,44,320,200,$fff8,3,8,32,320,320   ;set up a slice
Show 0                           ;show the bitmap
MouseWait
```

See Also:

Use Slice, Show, FreeSlices

Statement: **Use Slice**

Syntax: Use Slice *Slice#*

Modes: Amiga/Blitz

Description:

Use Slice is used to set the specified slice object as being the currently used slice. This is required for commands such as **Show**, **ShowF**, **ShowB** and Blitz mode **RGB**.

Example:

```
;
; program example
;
BLITZ                                          ;into blitz mode...
BitMap 0,320,200,3                             ;set up a bitmap
Circlef 160,100,80,2                           ;draw a circle on it
Slice 0,44,320,100,$fff8,3,8,8,320,320     ;one slice...
Slice 1,44,320,146,$fff8,3,8,8,320,320     ;another...
Use Slice 0                                    ;use the first one..
Show 0
RGB 2,15,15,0                                  ;Rgb/Show affects slice 0
Use Slice 1                                    ;use slice 1
Show 0
RGB 2,0,8,15                                   ;Rgb and Show into it
MouseWait
```

See Also:

Slice, FreeSlices

Statement: **FreeSlices**

Syntax: FreeSlices

Modes: Amiga/Blitz

Description:

FreeSlices is used to completely 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.

See Also:

Slice

Statement: **Show**

Syntax: Show *Bitmap#[,X,Y]*

Modes: Amiga/Blitz

Description:

Show is 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 it's 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.

Example:

```
;
; scrolling bitmap program example
;
BLITZ                         ;Go into Blitz Mode
BitMap 0,640,200,2             ;create bitmap 0
Circlef 320,100,80,1           ;draw a cricle on it..
Circlef 320,100,40,2           ;and another...
Slice 0,44,320,200,$fff8,2,8,4,640,640   ;create slice 0

For k=0 To 319               ;start of loop
  VWait                        ;wait for top of frame
  Show 0,k,0                   ;show bitmap 0
Next                          ;end of loop

MouseWait
```

See Also:

ShowF, ShowB

Statement: **ShowF**

Syntax: ShowF *BitMap#[,X,Y[,ShowB X]]*

Modes: Amiga/Blitz

Description:

ShowF is 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 it's top-left. Omitting the *X* and *Y* parameters is identical to supplying *X* and *Y* 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.

Example:

```
;
; dpf slice example program example
;
BLITZ                        ;blitz mode
BitMap 0,640,200,2                ;create a bitmap
Circlef 320,100,80,1              ;put a circle on it
Circlef 320,100,40,2              ;and another
Slice 0,44,320,200,$fffa,4,8,32,640,640   ;dual-playfield slice!
ShowB 0,160,0                     ;show background bitmap

For k=0 To 319              ;begin a loop
  VWait                       ;wait for vertical blank
  ShowF 0,k,0                 ;show foreground bitmap
Next                          ;end of loop
```

```
MouseWait
```

Statement: **ShowB**

Syntax: ShowB *BitMap#[,X,Y[,ShowF X]]*

Modes: Amiga/Blitz

Description:

ShowB is 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 it's top-left. Omitting the *X* and *Y* parameters is identical to supplying *X* and *Y* 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.

Example:

```
;
; showb and showf program example
;
BLITZ                          ;blitz mode
BitMap 0,640,200,2                   ;create a bitmap
Circlef 320,100,80,1               ;put a circle on it
Circlef 320,100,40,2               ;and another
Slice 0,44,320,200,$fffa,4,8,32,640,640    ;dual-playfield slice!
ShowF 0,160,0                      ;show background bitmap

For k=0 To 319                    ;begin a loop
 VWait                          ;wait for vertical blank
 ShowB 0,k,0                      ;show foreground bitmap
Next                           ;end of loop

MouseWait
```

Statement: **ColSplit**

Syntax: ColSplit *Colour Register,Red,Green,Blue,Y*

Modes: Amiga/Blitz

Description:

ColSplit 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.

Example:

```
;
; colsplit program example
;
BLITZ                              ;enter blitz mode
BitMap 0,320,200,1                 ;get an empty bitmap
Slice 0,44,320,200,$fff8,1,8,32,320,320   ;set up a slice
Show 0                             ;show the bitmap

For k=0 To 15                      ;begin loop
 ColSplit 0,k,k,k,k*17             ;set background register at
                                   ;a clever Y position
Next                               ;end loop

MouseWait
```

Statement: **CustomCop**

Syntax: CustomCop *Copin$,Y*

Modes: Amiga/Blitz

Description:

CustomCop 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.

Example:

```
;
; custom copper list program example
;
BLITZ                              ;Blitz mode
#BPLMOD1=$108                      ;some clever stuff..
#BPLMOD2=$10A                      ;ditto
BitMap 0,320,400,3

For k=7 To 1 Step -1
 Circlef 160,250,k*10,k            ;draw the SUN!
Next

Slice 0,44,320,200,$fff8,3,8,32,320,320   ;set up a slice
RGB 0,0,8,15

For k=1 To 7
 RGB k,15,k*2,0
Next

ColSplit 0,0,0,8,150               ;groovy colour split
co$=Mki$(#BPLMOD1)+Mki$(-122)
co$+Mki$(#BPLMOD2)+Mki$(-122)
CustomCop co$,150+44               ;custom copper instructions
```

```
For k=0 To 199
 VWait
 Show 0,0,k                     ;up comes the sun...
Next

MouseWait
```

See Also:

ColSplit

Statement: **ShowBlitz**

Syntax: **ShowBlitz**

Modes: Blitz

Description:

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.

Function: **CopLoc**

Syntax: **CopLoc**

Modes: Amiga/Blitz

Description:

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.

Example:

```
;
; print out of copper list program example
;
Slice 0,44,3

For k=0 To CopLen-1 Step 4
 NPrint Hex$(k),":",Hex$(Peek.l(CopLoc+k))
Next

MouseWait
```

See Also:

CopLen

Function: **CopLen**

Syntax: CopLen

Modes: Amiga/Blitz

Description:

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.

See Also:

CopLoc

Statement: **Display**

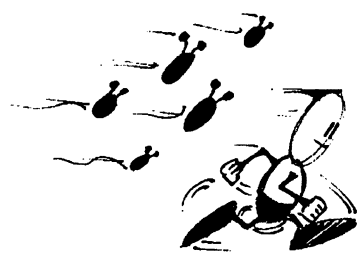
Syntax: Display *On| Off*

Modes: Blitz

Description:

Display 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.

20. Sprites



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 it's 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.

However, all this power does not come cheap. There are some limitations that must be observed when using sprites:

* In this release of Blitz 2, sprites are only available in Blitz mode.

* Sprites must be of either 3 or 15 colours (2 or 4 bitplanes).

* Each Blitz mode slice may display a maximum of up to 8 sprites. Other conditions may lower this maximum.

* Sprites are always displayed in low resolution mode, regardless of the display mode of the slice they are in.

* Sprites may only be positioned at low resolution pixel positions.

Sprites are displayed through the use of eight 'sprite channels', numbered 0 through 7. To display a sprite, you tell a sprite channel to display a specific image at a specific position. If you are displaying a three colour sprite, you may specify any of the eight sprite channels (0 through 7).

If you are displaying a fifteen colour sprite, you may only specify an even-numbered sprite channel (0,2,4,6). Fifteen colour sprites also 'tie-up' the associated odd-numbered sprite channel. For example, displaying a fifteen colour sprite through sprite channel 2 will make sprite channel 3 unavailable. This is because each 15 colour sprites requires 2 sprite channels.

The Amiga's hardware actually limits individual sprites to a maximum width of 16 lo-res pixels. However, Blitz 2 allows you to display sprites of greater width by splitting a shape up into groups of sixteen pixels. This means that a Blitz 2 'sprite' may take up more than one sprite channel. For example, a 32 pixel wide 3 colour 'sprite' displayed through sprite channel 4 will actually be converted to two 16 pixel wide sprites displayed through channels 4 and 5. Similarly, a 48 pixel wide 15 colour 'sprite' displayed through sprite channel 0 will take up sprite channels 0 through 5.

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

Statement: **GetaSprite**

Syntax: GetaSprite *Sprite#,Shape#*

Modes: Amiga/Blitz

Description:

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 2 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 anymore. In this case, it is best to free up the shape object (using **Free Shape**) to conserve as much valuable chip memory as possible.

Example:

```
;
; simple sprites example
;
BitMap 0,320,DispHeight,2      ;create a bitmap
Boxf 0,0,63,63,1            ;draw some stuff on it..
Boxf 8,8,55,55,2
Boxf 16,16,47,47,3
GetaShape 0,0,0,64,64         ;turn stuff into a shape
GetaSprite 0,0              ;turn shape into a sprite
Free Shape 0                ;we don't need the shape anymore...
BLITZ                   ;go into blitz mode.
Cls                     ;clear bitmap
Slice 0,44,2                ;create a slice
Show 0                     ;show bitmap 0 in the slice
For k=0 To 1                ;Since the sprite is 64 pixels wide,
RGB k*4+17,15,15,0               ;it will require 4 sprite channels (64/16)
RGB k*4+18,15,8,0               ;therefore, we must set palette
RGB k*4+19,15,4,0               ;registers appropriately
Next

For k=0 To 319              ;start of loop
 VWait                   ;wait for vertical blank
 ShowSprite 0,k,100,0           ;show the sprite...
Next                    ;end of loop

MouseWait
```

See Also:

Free Sprite, ShowSprite

Statement: **ShowSprite**

Syntax: ShowSprite *Sprite#,X,Y,Sprite Channel*

Modes: Amiga/Blitz

Description:

ShowSprite is the command used to actually display a sprite through a sprite channel. *X* and *Y* specify the position the sprite is to be displayed at. These parameters are ALWAYS given in lo-resolution pixels. *Sprite Channel* is a value 0 through 7 which decides which sprite channel the sprite should be display through.

See Also:

GetaSprite

Statement: **InFront**

Syntax: InFront *Sprite Channel*

Modes: Amiga/Blitz

Description:

A feature of sprites is that they may be displayed either 'in front of' or 'behind' the bitmap graphics they are appearing in. The **InFront** command allows you to determine which sprites appear in front of bitmaps, and which sprites appear behind.

Sprite Channel 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 *Sprite Channel* will appear BEHIND any bitmap graphics. Sprites displayed through channels less than *Sprite Channel* 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. For dualplayfield slices, use **InFrontF** and **InFrontB**.

Example:

```
;
; sprite priorities example
;
BitMap 0,320,DispHeight,2     ;create a bitmap
Boxf 0,0,63,63,1          ;draw some stuff on it..
Boxf 8,8,55,55,2
Boxf 16,16,47,47,3
GetaShape 0,0,0,64,64       ;turn stuff into a shape
GetaSprite 0,0          ;turn shape into a sprite
Free Shape 0            ;we don't need the shape anymore...
BLITZ                ;go into blitz mode.
Cls                  ;clear bitmap
```

```
Slice 0,44,2             ;create a slice
Show 0                   ;show bitmap 0 in the slice

For k=0 To 3             ;This loop will set all 3 colour
  RGB k*4+17,15,15,0        ;sprites to the same colours...
  RGB k*4+18,15,8,0
  RGB k*4+19,15,4,0
Next

Circlef 0,160,100,90,3      ;a circle +...
Circlef 0,160,100,80,0      ;a hole = a donut!
InFront 4                ;sprites 4-7 are 'behind'

For k=0 To 319            ;start of loop
  VWait                  ;wait for vertical blank
  ShowSprite 0,k,20,0       ;show in front sprite...
  ShowSprite 0,k,120,4      ;show behind sprite
Next                     ;end of loop

MouseWait
```

See Also:

InFrontF, InFrontB

Statement: **InFrontF**

Syntax: InFrontF *Sprite Channel*

Modes: Amiga/Blitz

Description:

InFrontF is 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, some between the playfields, and some behind both playfields.

Please refer to **InFront** for more information on the *Sprite Channel* parameter.

Example:

```
;
; sprites example
;
BitMap 1,320,200,2         ;create 'background' bitmap
Boxf 80,50,240,150,3       ;draw a box on it for scenery
BitMap 0,320,200,2         ;create 'foreground' bitmap
Boxf 0,0,63,63,1          ;draw some boxes...Boxf 8,8,55,55,2          ;
Boxf 16,16,47,47,3
GetaShape 0,0,0,64,64        ;pick up a shape
GetaSprite 0,0            ;turn it into a sprite
Free Shape 0              ;free shape as we no longer need it
Cls                    ;clear bitmap
```

```
Circlef 160,100,90,3          ;make some foreground scenery
Circlef 160,100,80,0
BLITZ                         ;go into BLITZ mode
Slice 0,44,320,200,$fff2,4,8,32,320,320 ;a dualplayfield slice!
ShowF 0                       ;show foreground bitmap
ShowB 1                       ;show background bitmap

For k=0 To 3                  ;set all sprite colours...
 RGB k*4+17,15,15,0
 RGB k*4+18,15,8,0
 RGB k*4+19,15,4,0
Next

InFrontF 0                    ;foreground is in front of sprites 2-7
InFrontB 4                    ;background is in front of sprites 4-7

For x=0 To 319        ;loop for sprite move
 VWait              ;wait for vertical blank
 ShowSprite 0,x,20,0 ;sprite behind foreground, infront of background
 ShowSprite 0,x,120,4 ;show sprite behind everything
Next                ;end of sprite move loop

MouseWait
```

See Also:

InFront, InFrontB

Statement: **InFrontB**

Syntax: InFrontB *Sprite Channel*

Modes: Amiga/Blitz

Description:

InFrontB is used on dualplayfield slices to determine sprite/playfield priority with respect to the background playfield. Using combinations of **InFrontB** and **InFrontF** (used for 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.

Please refer to **InFront** for more information on the *Sprite Channel* parameter.

See Also:

InFront, InFrontF

Statement: **LoadSprites**

Syntax: LoadSprites *Sprite#[,Sprite#],Filename$*

Modes: Amiga

Description:

LoadSprites 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.

See Also:

SaveSprites

Statement: **SaveSprites**

Syntax: SaveSprites *Sprite#,Sprite#,Filename$*

Modes: Amiga

Description:

SaveSprites 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.

See Also:

LoadSprites

21. Blitting

This section will cover all commands which allow you to draw shapes onto bitmaps using the Amiga's 'blitter' chip.

Statement: **Blit**

Syntax: **Blit** *Shape#,X,Y[,Excessonoff]*

Modes: Amiga/Blitz

Description:

Blit is 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.

Example:

```
;
; getashape and Blit example
;
Screen 0,3
ScreensBitMap 0,0
Cls
Circlef 32,32,32,3
Circlef 32,32,16,2
GetaShape 0,0,0,64,64
Cls
Blit 0,160,100
MouseWait
```

See Also

BlitMode, **QBlit**, **BBlit**

Statement: **BlitMode**

Syntax: **BlitMode** *BLTCON0*

Modes: Amiga/Blitz

Description:

The **BlitMode** command 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 and their effects:

BLTCON0 Mode	Effect
CookieMode	Shapes are drawn 'as is'.
EraseMode	An area the size and shape of the shape willbe 'erased' on the destination bitmap.
InvMode	An area the size and shape of the shape willbe 'inversed' 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 2 assume the following blitter channel setup:

Channel	Use
A	Pointer to shape's cookie cut
B	Pointer to shape data
C	Pointer to destination
D	Pointer to destination

Example:

```
;
; different blitmode examples
;
Screen 0,3                  ;open an intuition screen
ScreensBitMap 0,0                 ;and use it's bitmap
Cls                     ;clear bitmap

For k=7 To 1 Step -1         ;start of loop
 Circlef 32,32,k*4,k         ;groovy circles
```

```
Next                    ;end of loop

GetaShape 0,0,0,64,64      ;pick shape up
Cls 2                    ;clear bitmap again, with colour 2
Circlef 160,100,120,90,6    ;draw a circle.
BlitMode CookieMode          ;try a blit mode
Blit 0,0,0
BlitMode EraseMode          ;another...
Blit 0,160,0
BlitMode InvMode            ;another...
Blit 0,0,100
BlitMode SolidMode          ;and a last...
Blit 0,160,100

MouseWait
```

See Also:

QBlitMode, BBlitMode, SBlitMode

Function: **CookieMode**

Syntax: **CookieMode**

Modes: Amiga/Blitz

Description:

The **CookieMode** function returns a value which may be used by one of the commands involved in blitting modes.

Using **CookieMode** as a blitting mode will cause a shape to be blitted cleanly, 'as is', onto a bitmap.

See Also:

BlitMode, BBlitMode, QBlitMode, SBlitMode, EraseMode, InvMode, SolidMode

Function: **EraseMode**

Syntax: **EraseMode**

Modes: Amiga/Blitz

Description:

The **EraseMode** 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.

See Also:

BlitMode, BBlitMode, QBlitMode, SBlitMode, CookieMode, InvMode, SolidMode

Statement: **InvMode**

Syntax: **InvMode**

Modes: Amiga/Blitz

Description:

The **InvMode** 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.

See Also:

BlitMode, BBlitMode, QBlitMode, SBlitMode, CookieMode, EraseMode, SolidMode

Statement: **SolidMode**

Syntax: **SolidMode**

Modes: Amiga/Blitz

Description:

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.

See Also:

BlitMode, BBlitMode, QBlitMode, SBlitMode, CookieMode, EraseMode, InvMode

Statement: **Queue**

Syntax: **Queue** *Queue#,Max Items*

Modes: Amiga/Blitz

Description:

The **Queue** command creates a queue object for use with the **QBlit** and **UnQueue** commands. What is a queue? Well, queues (in the Blitz 2 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 it's new location on the display
Step 5: If there are any more objects to move, go 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. Lets have a 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 it's 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 2's queue system. The **UnQueue** command performs step 1, and the **QBlit** command performs step 4.

Queues purpose is to initialize 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 *Max Items* parameter.

Example:

```
;
; queue and unqueue blitting example
;
Screen 0,1              ;open intuition screen
ScreensBitMap 0,0       ;use it's bitmap
Cls                     ;clear the bitmap
Circlef 16,16,16,1      ;draw a circle
GetaShape 0,0,0,32,32   ;turn it into a shape
Cls                     ;clear the screen again
Queue 0,8               ;initialized our queue - 8 items max!
BLITZ                   ;go into blitz mode for speed!

For y=0 To 160          ;move down the bitmap
  VWait                 ;wait for top of frame
  UnQueue 0             ;erase all previously QBlitted items
  For x=1 To 8          ;move across the bitmap
    QBlit 0,0,x*32,y    ;draw object and remember it in queue 0
  Next                  ;again...
Next                    ;again...

MouseWait
```

See Also:

QBlit, UnQueue

Statement: **QBlit**

Syntax: **QBlit** *Queue#,Shape#,X,Y[,Excessonoff]*

Modes: Amiga/Blitz

Description:

QBlit 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.

Please refer to the **Queue** command for an explanation of the use of queues.

The optional *Excessonoff* parameter works identically to the *Excessonoff* parameter used by the **Blit** command. Please refer to the **Blit** command for more information on this parameter.

See Also:

Queue, UnQueue, Blit

Statement: **UnQueue**

Syntax: **UnQueue** *Queue#[,BitMap#]*

Modes: Amiga/Blitz

Description:

UnQueue is used to erase all 'remembered' items in a queue. Items are placed in a queue by use of the **QBlit** command. Please refer to **Queue** for a full explanation of queues and their usage.

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.

Example:

```
;
; unqueueing from separate bitmap
;
Screen 0,1                    ;open intuition screen
ScreensBitMap 0,0                ;use it's bitmap
Cls                        ;clear the bitmap
Circlef 16,16,16,1                ;draw a circle
GetaShape 0,0,0,32,32                ;turn it into a shape
Cls                        ;clear the screen again

For k=1 To 100
  Circlef Rnd(320),Rnd(DispHeight),Rnd(50),1     ;draw some circles
Next
```

```
CopyBitMap 0,1          ;make an identical copy of bitmap 0
Queue 0,8               ;initialized our queue - 8 items max!
BLITZ                   ;go into blitz mode for speed!

For y=0 To 160          ;move down the bitmap
 VWait                  ;wait for top of frame
 UnQueue 0,1            ;erase all previously QBlitted items
 For x=1 To 8           ;move across the bitmap
  QBlit 0,0,x*32,y      ;draw object and remember it in queue 0
 Next                   ;again...
Next                    ;again...

MouseWait
```

Statement: **FlushQueue**

Syntax: **FlushQueue** *Queue#*

Modes: Amiga/Blitz

Description:

FlushQueue will force the specified queue object to be 'emptied', causing the next **UnQueue** command to have no effect.

See Also:

Queue, QBlit

Statement: **QBlitMode**

Syntax: **QBlitMode** *BLTCON0*

Modes: Amiga/Blitz

Description:

QBlitMode allows you to control how the blitter operates when **QBlit**ting shapes to bitmaps. Please refer to **BlitMode** for more information on this command.

See Also:

BlitMode

Statement: **Buffer**

Syntax: **Buffer** *Buffer#,Memorylen*

Modes: Amiga/Blitz

Description:

The **Buffer** command is 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 the **Buffer** 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 **BBlit**ed, and the maximum number of shapes to be **BBlit**ed between **UnBuffer**s.

A *Memorylen* of 16384 should be plenty for most situations, but may need to be increased if you start getting 'Buffer Overflow' error messages.

Example:

```
;
; buffer blitting example
;
BitMap 0,64,64,1
Boxf 0,0,63,63,1
GetaShape 0,0,0,64,64
FindScreen 0
ScreensBitMap 0,0
Buffer 0,16384        ;16384 bytes for buffer

For x=0 To 600
 VWait
 UnBuffer 0        ;undo eny blits
 BBlit 0,0,x,192   ;buffer blit
Next

MouseWait
```

Statement: **BBlit**

Syntax: **BBlit** *Buffer#,Shape#,X,Y[,Excessonoff]*

Modes: Amiga/Blitz

Description:

The **BBlit** command is used to draw a shape onto the currently used bitmap, and preserve the overwritten area into a previously initialized buffer. For more information on how buffers work, please refer to the **Buffer** command.

The optional *Excessonoff* parameter works identically to the *Excessonoff* parameter used by the **Blit** command. Please refer to the **Blit** command for more information on this parameter.

Example:

```
;
; buffer blitting example
;
Screen 0,3                    ;open intuition screen
ScreensBitMap 0,0             ;use it's bitmap for our graphics
Cls                           ;clear the bitmap
Circlef 8,8,8,7               ;draw a circle
GetaShape 0,0,0,20,16         ;get it for use as a shape
Cls                           ;clear bitmap again

For k=1 To 100                ;draw 100 random box's
  Boxf Rnd(320),Rnd(200),Rnd(320),Rnd(200),Rnd(6)+1
Next

Buffer 0,16384                ;set buffer memory size

While Joyb(0)=0               ;loop into mouse button clicked
  VWait                       ;wait for vertical blank
  UnBuffer 0                  ;replace areas on bitmap
  BBlit 0,0,SMouseX/2+80,SMouseY/2+50  ;blit object - add to buffer
Wend
```

See Also:

Buffer, UnBuffer

Statement: **UnBuffer**

Syntax: **UnBuffer** *Buffer#*

Modes: Amiga/Blitz

Description:

UnBuffer is 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.

See Also:

Buffer, BBlit

Statement: **FlushBuffer**

Syntax: **FlushBuffer** *Buffer#*

Modes: Amiga/Blitz

Description:

FlushBuffer will force the specified buffer object to be 'emptied', causing the next **UnBuffer** command to have no effect.

See Also:

Buffer, BBlit

Statement: BBlitMode

Syntax: **BBlitmode** *BLTCON0*

Modes: Amiga/Blitz

Description:

BBlitMode allows you to control how the blitter operates when **BBlit**ting shapes to bitmaps. Please refer to **BlitMode** for more information on this command.

See Also:

BlitMode

Statement: Stencil

Syntax: **Stencil** *Stencil#,BitMap#*

Modes: Amiga/Blitz

Description:

The **Stencil** command will create 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.

Example:

```
;
; stencil blit examples
;
For k=1 To 7             ;draw some concentric circles
 Circle 160,115,k*10,k
Next

Stencil 0,0              ;make a stencil out of bitmap 0
Buffer 0,16384           ;set up a buffer for BBlit
BLITZ                    ;into Blitz mode!

For x=0 To 280           ;move shapes across...
 VWait                   ;wait for vertical blank
 UnBuffer 0              ;replace BBlits
 For y=50 To 150 Step 50
  BBlit 0,0,x,y          ;BBlit some of our shapes
 Next
 ShowStencil 0,0         ;replace stencil area
```

```
Next

MouseWait
```

Statement: SBlit

Syntax: **SBlit** *Stencil#,Shape#,X,Y[,Excessonoff]*

Modes: Amiga/Blitz

Description:

SBlit works identically to the **Blit** command, and also updates the specified *Stencil#*. This is an easy way to render 'foreground' graphics to a bitmap.

Example:

```
;
; more stencil blitting
;
Screen 0,3                  ;open an intuition screen
ScreensBitMap 0,0              ;find it's bitmap
Boxf 0,0,31,31,3            ;draw a box on the bitmap
GetaShape 0,0,0,32,32          ;pick it up as shape 0
Cls                       ;clear bitmap
Boxf 0,0,15,15,4            ;draw another box
GetaShape 1,0,0,16,16          ;pick it up as shape 1
Cls                       ;another cls
Stencil 0,0                 ;create a stencil

For k=7 To 1 Step -1        ;draw a background 'bullseye'
 Circlef 160,115,k*10,k
Next

For k=1 To 50             ;draw up 50 random 'foreground' blocks
 SBlit 0,1,Rnd(320-16),Rnd(200-16)
Next

Buffer 0,16384          ;initialize buffer

BLITZ                 ;into BLITZ MODE!

For x=0 To 280                ;start of loop
 VWait                    ;wait for vertical blank
 UnBuffer 0                 ;replace buffer contents
 For y=50 To 150 Step 50
  BBlit 0,0,x,y             ;blit up our shape
 Next
 ShowStencil 0,0                ;cover-up stenciled areas
Next

MouseWait
```

Statement: SBlitMode

Syntax: **SBlitMode** *BLTCON0*

Modes: Amiga/Blitz

Description:

SBlitmode is used to determine how the **SBlit** command operates. Please refer to the **BlitMode** command for more information on blitting modes.

See Also:

BlitMode

Statement: ShowStencil

Syntax: **ShowStencil** *Buffer#,Stencil#*

Modes: Amiga/Blitz

Description:

ShowStencil is 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.

In order to understand the following, it is recommended that the description of the **Buffer** command first be read, as stencils and buffers are closely connected.

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 **BBlit**ted, executing the **ShowStencil** command will repair the damage done to the foreground graphics!

Example:

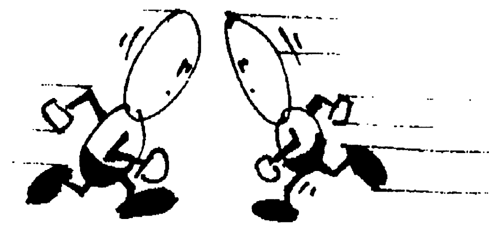
```
;
; bblits with stencils
;
Screen 0,3                    ;an intuition screen
ScreensBitMap 0,0                 ;it's bitmap...now ours
Cls                          ;clear bitmap
Boxf 0,0,7,15,1              ;draw a shape...
Boxf 8,6,15,11,2
GetaShape 0,0,0,16,16          ;pick it up as our shape.
Cls                     ;clear bitmap again
Boxf 80,50,240,150,3          ;draw some stuff...
Boxf 90,60,230,140,0
```

```
Box 85,55,235,145,0
Stencil 0,0                ;make a stencil out of the bitmap
Cls                        ;clear bitmap again
Circlef 160,100,90,4       ;draw background graphics...
Boxf 80,50,240,150,3       ;and foreground (again!)
Boxf 90,60,230,140,4
Box 85,55,235,145,4
Buffer 0,16384             ;set up a buffer for BBlit
BLITZ                      ;go into blitz mode for more speed

For x=0 To 300             ;start of loop
 VWait:UnBuffer 0          ;wait for top of frame; replace buffer
 For y=40 To 140 Step 50   ;start of loop to draw 3 shapes
  BBlit 0,0,x,y            ;put up a shape
 Next
 ShowStencil 0,0           ;replace foreground
Next

MouseWait
```

See Also:

Buffer, BBlit, Stencil, SBlit, UnBuffer

22. Collisions



This section deals with various commands involved in the detection of object collisions.

Statement: **SetColl**

Syntax: SetColl *Colour,Bitplanes[,Playfield]*

Modes: Amiga/Blitz

Description:

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 'collidable' colour. *Bitplanes* refers to the number of bitplanes (depth) of the bitmap collisions are to be tested for 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.

Example:

```
;
; death star collision example
;
BitMap 0,320,200,4          ;create a 16 colour bitmap
BitMapOutput 0              ;send print statements there
Boxf 0,0,7,7,1             ;draw a box on the bitmap
GetaShape 0,0,0,8,8          ;pick it up as a shape
GetaSprite 0,0             ;turn shape into a sprite
Free Shape 0               ;free shape - we don't need it
Cls                      ;Clear the bitmap

BLITZ                      ;BLITZ MODE!
Slice 0,44,320,200,$fff8,4,8,32,320,320 ;simple slice
Show 0                      ;show bitmap in slice

For k=1 To 100                  ;draw 100 stars
  Plot Rnd(320),Rnd(200),Rnd(14)+1     ;in any colour but 15!
Next
```

```
Circlef 160,100,40,15   ;the death star! in colour 15!
SetColl 15,4            ;collide with colour 15
Mouse On                ;enable mouse
Pointer 0,0             ;set mouse pointer
While Joyb(0)=0         ;while the mouse button is left alone...
 VWait                  ;wait for vertical blank
 DoColl                 ;ask Blitz )( to suss collisions
 Locate 0,0             ;text cursor position
 If PColl(0)            ;did sprite channel 0 collide with bitmap ?
  Print "BANG!"         ;Yes - BANG!
 Else
  Print "     "         ;No
 EndIf
Wend
```

See Also:

SetCollOdd, SetCollHi, DoColl, PColl

Statement: **SetCollOdd**

Syntax: SetCollOdd

Modes: Amiga/Blitz

Description:

SetCollOdd is 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 (ie: 1,3,5...). This allows you to design bitmap graphics in such a way that even coloured areas are 'safe' (ie: they will not report a collision) whereas odd colour areas are 'unsafe' (ie: they will report a collision).

The **DoColl** and **PColl** commands are used to detect the actual sprite/bitmap collisions.

See Also:

SetColl, SetCollHi, DoColl, PColl

Statement: **SetCollHi**

Syntax: SetCollHi *BitPlanes*

Modes: Amiga/Blitz

Description:

SetCollHi 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.

See Also:

SetColl, SetCollOdd, DoColl, PColl

Statement: **DoColl**

Syntax: DoColl

Modes: Blitz

Description:

DoColl is 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.

See Also:

SetColl, **SetCollOdd**, **SetCollHi**, **PColl**

Function: **PColl**

Syntax: PCOll *(Sprite Channel)*

Modes: Blitz

Description:

The **PColl** function may be used to find out if a particular sprite has collided with any bitmaps. *Sprite Channel* refers to the sprite channel 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. Please refer to **DoColl** for more information.

See Also:

SetColl, **SetCollOdd**, **SetCollHi**, **DoColl**

Function: SColl

Syntax: SColl *(Sprite Channel, Sprite Channel)*

Modes: Blitz

Description:

SColl may be 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**.

See Also:

DoColl

Function: ShapesHit

Syntax: ShapesHit *(Shape#,X,Y,Shape#,X,Y)*

Modes: Amiga/Blitz

Description:

The **ShapesHit** 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).

See Also:

ShapeSpriteHit, **SpritesHit**

Function: ShapeSpriteHit

Syntax: ShapeSpriteHit *(Shape#,X,Y,Sprite#,X,Y)*

Modes: Amiga/Blitz

Description:

The ShapeSpriteHit 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.

See Also:

ShapesHit, **SpritesHit**

Function: **SpritesHit**

Syntax: SpritesHit *(Sprite#,X,Y,Sprite#,X,Y)*

Modes: Amiga/Blitz

Description:

The **SpritesHit** 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).

See Also:

ShapesHit, **ShapeSpriteHit**

Function: **RectsHit**

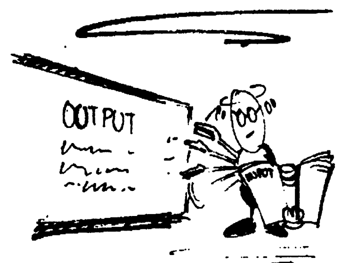
Syntax: RectsHit *(X1,Y1,Width1,Height1,X2,Y2,Width2,Height2)*

Modes: Amiga/Blitz

Description:

The **RectsHit** 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).

23. BlitzIO



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.

Statement: **BlitzKeys**

Syntax: **BlitzKeys** *On| Off*

Modes: Blitz

Description:

BlitzKeys is 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.

Example:

```
;
; bitmap output with blitzkeys on program example
;
BLITZ
BitMap 0,320,DispHeight,3
BitMapOutput 0
Slice 0,44,3
Show 0
BlitzKeys On
NPrint "Type Away..... (Click mouse to exit)"

While Joyb(0)=0
 Print Inkey$
Wend

End
```

See Also:

BlitzRepeat

Statement: **BlitzRepeat**

Syntax: **BlitzRepeat** *Delay,Speed*

Modes: Blitz

Description:

BlitzRepeat 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 will the Blitz mode keyboard reading is enabled. This is done using the **BlitzKeys** command.

See Also:

BlitzKeys

Function: **RawStatus**

Syntax: RawStatus *(Rawkey)*

Modes: Blitz

Description:

The **RawStatus** 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.

Example:

```
;
; rawkey program example
;
BLITZ
BitMap 0,320,DispHeight,3
BitMapOutput 0
Slice 0,44,3
Show 0
BlitzKeys On
NPrint "Click Mouse to exit..."

While Joyb(0)=0

 Locate 0,1
 Print "F1 Key is Currently:"
 If RawStatus(80)
  Print "Down"
 Else
  Print "Up  "
 EndIf
Wend

End
```

Statement: **Mouse**

Syntax: **Mouse** *On/ Off*

Modes: Amiga

Description:

The **Mouse** command turns on or off Blitz mode's ability to read the mouse. Once a **Mouse** *On* has been executed, programs can read the mouse's position or speed in Blitz mode.

Example:

```
;
; blitz mouse program example
;
BLITZ
BitMap 0,320,DispHeight,3
Slice 0,44,3
Show 0
Mouse On

While Joyb(0)=0
 Line 160,100,MouseX,MouseY,1
Wend

End
```

Statement: **Pointer**

Syntax: **Pointer** *Sprite#,Sprite Channel*

Modes: Blitz

Description:

The **Pointer** command 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 the 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.

Finally, **Pointer** is executed to attach the Sprite.

Example:

```
;
; custom pointer program example
;
LoadShape 0,"MySprite"          ;Must be 4 or 16 colour shape
LoadPalette 0,"MySprite",16      ;pick up palette
GetaSprite 0,0                 ;make shape into sprite!
```

```
BLITZ                           ;go into BLITZ MODE!
BitMap 0,320,DispHeight,3           ;set up a bitmap
Slice 0,44,3                      ;turn on slice
Use Palette 0                   ;add sprites palette to slice
Show 0                            ;show bitmap
Mouse On                            ;turn pon blitz mode mouse reading
Pointer 0,0                       ;attach pointer
MouseWait
```

See Also:

Mouse

Statement: **MouseArea**

Syntax: **MouseArea** *Minx,Miny,Maxx,Maxy*

Modes: Blitz

Description:

MouseArea 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.

Example:

```
;
; mouse area program example
;
LoadShape 0,"MySprite"          ;Must be 4 or 16 colour shape
LoadPalette 0,"MySprite",16     ;get the sprites palette
GetaSprite 0,0             ;turn shape into a sprite
BLITZ                      ;go into blitz mode
BitMap 0,320,DispHeight,3       ;initialize a bitmap
Slice 0,44,3                  ;create a slice
Use Palette 0              ;add sprites colours to slice
Show 0                        ;show bitmap
Mouse On                        ;turn mouse on
MouseArea 80,50,240,150 ;limit mouse to 'middle' area of display
Pointer 0,0                   ;attach pointer
MouseWait
```

Function: **MouseX**

Syntax: **MouseX**

Modes: Blitz

Description:

If Blitz mode mouse reading has been enabled using a **Mouse** *On* command, the **MouseX** function may be using 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.

Example:

```
;
; pretty lines program example
;
BLITZ                          ;into blitz mode
BitMap 0,320,DispHeight,3           ;make a bitmap
Slice 0,44,3                   ;and a slice
Show 0                         ;show bitmap in slice
While NOT Joyb(0)                    ;while LMB not pushed...
 VWait                         ;wait for vertical blank
 Line 160,100,MouseX,MouseY,Rnd(7)+1     ;pretty lines
Wend
```

See Also:

MouseY, MouseXSpeed, MouseYSpeed

Function: **MouseY**

Syntax: MouseY

Modes: Blitz

Description:

If Blitz mode mouse reading has been enabled using a **Mouse** *On* command, the **MouseY** function may be using 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.

See Also:

MouseX, **MouseXSpeed**, **MouseYSpeed**

Function: **MouseXSpeed**

Syntax: MouseXSpeed

Modes: Blitz

Description:

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 interupt.

See Also:

MouseX, MouseY, MouseYSpeed

Function: **MouseYSpeed**

Syntax: MouseYSpeed

Modes: Blitz

Description:

If Blitz mode mouse reading has been enabled using a **Mouse** *On* command, the **MouseYSpeed** function may be using 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 interupt.

See Also:

MouseX, MouseY, MouseXSpeed

Statement: **LoadBlitzFont**

Syntax: LoadBlitzFont *BlitzFont#,Fontname.font$*

Modes: Amiga

Description:

LoadBlitzFont 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 **LoadBlitzFont** 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 eight by eight non-proportional fonts.

Example:

```
;
; blitzfont program example
;
LoadBlitzFont 0,"Myfont.font"     ;load blitzfont #0
Screen 0,3                        ;open a screen
```

```
ScreensBitMap 0,0              ;get the screens bitmap
BitMapOutput 0                 ;send Print to bitmap...
Print "Hello - this is my font"  ;do a Print
MouseWait
```

See Also:

Use BlitzFont, Free BlitzFont, BitMapOutput

Statement: **Use BlitzFont**

Syntax: Use BlitzFont *BlitzFont#*

Modes: Amiga/Blitz

Description:

If you have loaded two or more blitzfont objects using **LoadBlitzFont**, **Use BlitzFont** may be used to select one of these fonts for future bitmap output.

Example:

```
;
; use blitzfont program example
;
LoadBlitzFont 0,"MyFont1.font"  ;load in a blitzfont...
LoadBlitzFont 1,"MyFont2.font"  ;and another...
Screen 0,3                      ;open a screen
ScreensBitMap 0,0               ;get bitmap of screen
BitMapOutput 0                  ;send 'Print' there...
Use BlitzFont 0                 ;use first blitzfont...
NPrint "This is My Font 1..."   ;print something
Use BlitzFont 1                 ;use second blitzfont...
NPrint "And this is My Font 2!" ;print something
MouseWait
```

See Also:

LoadBlitzFont, Free BlitzFont

Statement: **Free BlitzFont**

Syntax: Free BlitzFont *BlitzFont#*

Modes: Amiga/Blitz

Description:

Free BlitzFont 'unloads' a previously loaded blitzfont object. This frees up any memory occupied by the font.

See Also:

LoadBlitzFont, Use BlitzFont

Statement: **BitMapOutput**

Syntax: BitMapOutput *BitMap#*

Modes: Amiga/Blitz

Description:

BitMapOutput 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 eight by eight non-proportional fonts.

BitMapOutput is mainly of use in Blitz mode, as other forms of character output become unavailable in Blitz mode.

Example:

```
;
; bitmapoutput program example
;
Screen 0,3                        ;open an Intuition screen
ScreensBitMap 0,0                     ;get it's bitmap
BitMapOutput 0                  ;send Print statements there...
Print "Printing on a bitmap!"  ;print something!
MouseWait
```

See Also:

LoadBlitzFont, Locate

Statement: **Colour**

Syntax: Colour *Foreground Colour[,Background Colour]*

Modes: Amiga/Blitz

Description:

Colour allows you to alter the colours use to render text to bitmaps. *Foreground colour* allows you to specify the colour text is rendered in, and the optional *Background colour* 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.

Example:

```
;
; colourful program example
;
Screen 0,3              ;open an Intuition screen
ScreensBitMap 0,0          ;use it's bitmap
BitMapOutput 0            ;send Print statements

Locate 0,2

For k=0 To 7             ;loop 1...
 For J=0 To 7            ;loop 2...
  If k<>j              ;some trickery...
   Colour k,j
   Print "* "
  EndIf
 Next
Next

MouseWait
```

See Also:

BitMapOutput

Statement: **Locate**

Syntax: Locate *X,Y*

Modes: Amiga/Blitz

Description:

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 eigth.

Y specifies a character position down the bitmap, and may be a fractional value. For example, a *Y* of 1.5 will set a cusor position one and a half characters down from the top of the bitmap.

Each bitmap maintains it's own cursor position. The **Locate** statement alters the cursor position of the bitmap specified in the most recently executed **BitMapOutput** statement.

Example:

```
;
; more colour program example
;
Screen 0,3              ;open an Intuition screen
ScreensBitMap 0,0          ;borrow it's bitmap
BitMapOutput 0            ;send print statements to bitmap 0

For k=1 To 100             ;start of loop...
```

```
  Locate Rnd(40),Rnd(DispHeight/8-7)  ;random cursor position
  Colour Rnd(7)+1                     ;random colour
  Print "*"                           ;print a 'star'
Next                                  ;end of loop...

MouseWait
```

See Also:

BitMapOutput, CursX, CursY

Function: **Cursx**

Syntax: CursX

Modes: Amiga/Blitz

Description:

When using **BitMapOutput** to render text to a bitmap, **CursX** may be used to find the horizontal character position at which the next character **Print**ed will appear.
CursX will reflect the cursor position of the bitmap specified in the most recently executed **BitMapOutput** statement.

Example:

```
;
; cursx program example
;
Screen 0,3              ;open an Intuition screen
ScreensBitMap 0,0       ;find it's bitmap
BitMapOutput 0          ;send Print statements there...
Locate 0,2              ;position bitmap cursor

For k=1 To 16           ;start a loop...

  While k>CursX         ;some trickery!
    Print "*"
  Wend

  NPrint ""             ;print a newline

Next

MouseWait
```

See Also:

BitMapOutput, CursY, Locate

Statement: **CursY**

Syntax: CursY

Modes: Amiga/Blitz

Description:

When using **BitMapOutput** to render text to a bitmap, **CursY** may be used to find the vertical character position at which the next character **Print**ed will appear.

CursY will reflect the cursor position of the bitmap specified in the most recently executed **BitMapOutput** statement.

See Also:

BitMapOutput, CursX, Locate

Statement: **BitMapInput**

Syntax: BitMapInput

Modes: Blitz

Description:

BitMapInput 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**. A **BitMapOutput** must also be executed before any **Edit$** or **Edit** commands are encountered.

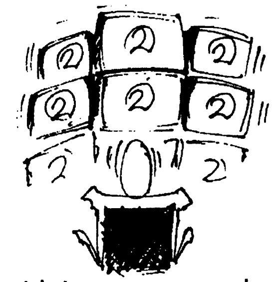
Example:

```
;
; bitmap input program example
;
Screen 0,3                    ;open an Intuition screen
ScreensBitMap 0,0             ;find it's bitmap
BitMapOutput 0                ;send Print statements there
BLITZ                         ;go into the infamous BLITZ mode!
BlitzKeys On                  ;turn on blitz mode keyboard reading.
BitMapInput                   ;get input from bitmap
Locate 0,2                    ;position cursor
a$=Edit$("Type Something!",40)  ;get some input
```

See Also:

BitMapOutput, BlitzKeys

24. Screens



The following section covers the Blitz 2 commands that let you open and control Intuition based Screen objects.

Statement: **Screen**

Syntax: **Screen** *Screen#,Mode[,Title$]*
or **Screen** *Screen#,X,Y,Width,Height,Depth,Viewmode,Title$,Dpen,Bpen[,BitMap#]*

Modes: Amiga

Description:

Screen will open an Intuition screen. The are 2 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.

Statement: **ShowScreen**

Syntax: **ShowScreen** *Screen#*

Modes: Amiga

Description:

ShowScreen will cause the specified screen object to be moved to the front of the display.

Statement: **WbToScreen**

Syntax: **WbToScreen** *Screen#*

Modes: Amiga

Description:

WbToScreen will assign 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. It's main usage is to allow you to open windows on the Workbench screen.

After execution, the Workbench screen will become the currently used screen.

Example:

```
;
; open a window on the workbench example program
;
WBenchToFront_              ;actually an OS call!
WbToScreen 0                 ;pick up workbench screen!
;
Window 0,2,1,600,180,15,"A Window on the WorkBench screen",0,1
Print "Click the right mouse button to quit"
While Joyb(0)<>2:Wend
WBenchToBack_
```

See Also:

FindScreen

Statement: **FindScreen**

Syntax: **FindScreen** *Screen#[,Title$]*

Modes: Amiga

Description:

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.

Example:

```
;
; open a window on the front screen example program
;
FindScreen 0                                 ;get frontmost screen
Window 0,0,0,100,100,0,"Our window",0,1        ;open window
MouseWait
```

See Also:

WBToScreen

Statement: **LoadScreen**

Syntax: LoadScreen *Screen#,Filename$[,Palette#]*

Modes: Amiga

Description:

LoadScreen 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 **Use Palette** statement.

Example:

```
;
; loadscreen example program
;
Screen 0,3,"Click LMB to quit"                ;open an intuition screen
LoadScreen 0,"TestScreen320x200x3",0         ;load an IFF ILBM pic.
Use Palette 0                                ;use it's palette
MouseWait
```

See Also:

SaveScreen

Statement: **SaveScreen**

Syntax: SaveScreen *Screen#,Filename$*

Modes: Amiga

Description:

SaveScreen will save 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$.*

Example:

```
;
; draw, save and then load screen example program
;
Screen 0,3                 ;open Intuition screen.
ScreensBitMap 0,0          ;pinch it's bitmap
BitMapOutput 0             ;send Print statements to screen's bitmap
Print "Draw on screen with LMB"
Print "Press RMB to save picture as file RAM:picture"
```

```
While JB<>2                    ;wait for RMB
 JB=Joyb(0)
 If JB=1 Then Plot SMouseX,SMouseY,2
Wend

Print "Saving the screen"
SaveScreen 0,"ram:picture"        ;save the screen
Cls                             ;clear bitmap (will affect screen)
Print "Press LMB to load it back in"
MouseWait
LoadScreen 0,"ram:picture",0      ;load back in.
Print "Press LMB to quit"
MouseWait
```

See Also:

LoadScreen

Function: SMouseX

Syntax: SMouseX

Modes: Amiga

Description:

SMouseX returns the horizontal position of the mouse relative to the left edge of the currently used screen.

Example:

```
;
; smousex&y program example program
;
Screen 0,2                    ;open a simple screen
ScreensBitMap 0,0             ;grab it's bitmap
BitMapOutput 0                ;send Print to bitmap
Print "Click LMB to quit"

While Joyb(0)=0               ;while no Mouse buttons pressed...
 Locate 0,1                   ;position bitmap cursor
 Print SMouseX," ",SMouseY    ;print X&Y of mouse
Wend
```

See Also:

SMouseY;

Function: **SMouseY**

Syntax: SMouseY

Modes: Amiga

Description:

SMouseY returns the vertical position of the mouse relative to the top of the current screen.

See Also:

SMouseX

Function: **ViewPort**

Syntax: **ViewPort(***Screen#***)**

Modes: Amiga

Description:

The **ViewPort** function returns the location of the specified screens ViewPort. The ViewPort address can be used with graphics.library commands and the like.

See Also:

RastPort

Statement: **ScreenPens**

Syntax: **ScreenPens** (*active text, inactive text, hilight, shadow, active fill, gadget fill*)

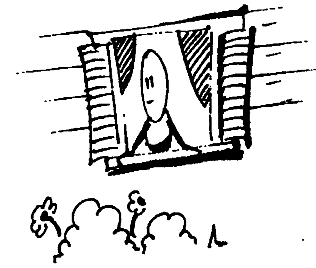
Modes: Amiga

Description:

ScreenPens configures the 10 default pens used for system gadgets in WorkBench 2.0. 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.

25. Windows



Windows are basically separate areas of a screen that are used for displaying information. These areas are independent, so if you write on one window, you will not write all over another, even if it is on top of the one you are writing on.

Windows must always appear within an Intution screen of some kind, be it the Workbench screen, or your own custom screen.

To efficiently handle windows, the following steps are recommended:

1) Set up a screen of some kind, using either **Screen** or **WBToScreen**

2) Open any windows you require on the screen

3) Use **WaitEvent** to detect any user activity in any of the windows

4) Decide what to do with the event, do it, then go back to step 3

Statement: **Window**

Syntax: **Window** *Window#,X,Y,Width,Height,Flags,Title$,Dpen,Bpen[,GadgetList#]*

Modes: Amiga

Description:

Window opens an Intuition window on the currently used screen. *Window#* is a unique object number for the new window. *X* and *Y* refer to the offset from the 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, dragbar 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 dragbar 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 the 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.

Window Flag	Value	Description
WINDOWSIZING	$0001	Attaches sizing gadget to bottom right corner of window and allows it to be sized.
WINDOWDRAG	$0002	Allows window to be dragged with the mouse by it's title bar.
WINDOWDEPTH	$0004	Lets windows be pushed behind or pulled in front of other windows.
WINDOWCLOSE	$0008	Attaches a closegadget to the upper left corner of the window.
SIZEBRIGHT	$0010	With GIMMEZEROZERO and WINDOWSIZING set, this will leave the right hand margin, the width of the sizing gadget, clear, and any drawing to the 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.
GIMMEZEROZERO	$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 convevient, 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.

Example:

```
;
; window on workbench example program
;
WbToScreen 0                 ;use workbench screen
WBenchToFront_               ;bring it to front.
Window 0,2,2,600,160,$1 I $2 I $4 I $8,"A Window",0,1
MouseWait
WBenchToBack_
```

Statement: **Use Window**

Syntax: **Use Window** *Window#*

Modes: Amiga

Description:

Use Window will cause the specified window object to become the currently used window. **Use Window** also automatically performs a **WindowInput** and **WindowOutput** on the specified window.

Example:

```
;
; use window example program
;
WBenchToFront_              ;From Intuition Library.
WbToScreen 0                ;Use Workbench as Screen #0.
Window 1,0,0,100,100,$f,"Window One",0,1
Window 2,100,100,100,100,$f,"Window Two",0,1    ;automatically 'used'
Print "This is in window two"
Use Window 0                ;use window 0
Print "This is in window one"
MouseWait
```

Statement: **Free Window**

Syntax: **Free Window** *Window#*

Modes: Amiga

Description:

Free 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.

Example:

```
;
; free window example program
;
WBenchToFront_              ;bring workbench screen to front of view.
WbToScreen 0                ;use workbench as screen 0
Window 0,0,0,300,100,$f,"Click to say bye bye",0,1
MouseWait
Free Window 0
```

Statement: **WindowInput**

Syntax: **WindowInput** *Window#*

Modes: Amiga/Blitz

Description:

WindowInput will cause 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 2 to get it's input from somewhere else useful (for example, using another **WindowInput** command) before executing another **Inkey$**, **Edit$** or **Edit** function.

See Also:

WindowOutput, Window, Use Window

Statement: **WindowOutput**

Syntax: **WindowOutput** *Window#*

Modes: Amiga/Blitz

Description:

WindowOutput will cause 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.

See Also:

WindowInput, Window, Use Window

Statement: **DefaultIDCMP**

Syntax: **DefaultIDCMP** *IDCMP_Flags*

Modes: Amiga

Description:

DefaultIDCMP 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 ('l ').

Any windows opened before any **DefaultIDCMP** command is executed will be opened using an IDCMP flags setting of: $2l $4l $8l $20l $40l $100l $200l $400l $40000l $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.

Here is a table of possible events and their IDCMP flags:

IDCMP Flag	Event
$2	Reported when a window has it's 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 occured.
$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'.

Example:

```
;
; simple idcmp example program
;
Screen 0,2                  ;simple screen

DefaultIDCMP $8             ;simple 'mouse buttons' IDCMP flag

Window 0,0,0,320,100,0,"Closes on mouseclick",0,1
Window 0,0,0,320,100,0,"Closes on mouseclick",0,1

DefaultIDCMP $400           ;simple 'key press' IDCMP flag

Window 1,0,100,320,100,0,"Closes on keypress",0,1

ev.l=WaitEvent

If ev=$8 Then Free Window 0 Else Free Window 1 ;close approprate window

WaitEvent
```

Statement: AddIDCMP

Syntax: **AddIDCMP** *IDCMP_Flags*

Modes: Amiga/Blitz

Description:

AddIDCMP 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.

Example:

```
;
; addidcmp example program
;
Screen 0,3
Window 0,0,0,320,DispHeight,$100f,"My Window",1,2

Repeat              ;repeat...
 ev.l=WaitEvent
 If ev=$10          ;has mmouse moved?
  If WCursY+8>=InnerHeight Then InnerCls:WLocate 0,0
  NPrint "Mouse moved!"
 EndIf
Until ev=512          ;until window closed
```

See Also:

DefaultIDCMP, **SubIDCMP**

Statement: **SubIDCMP**

Syntax: **SubIDCMP** *IDCMP_Flags*

Modes: Amiga/Blitz

Description:

SubIDCMP 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.

See Also:

DefaultIDCMP, **AddIDCMP**

Statement/Function: **WaitEvent**

Syntax: **WaitEvent**

Modes: Amiga

Description:

WaitEvent will halt program excution 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 occured.

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

Example:

```
;
; wait event example program
;
Screen 0,2                                    ;open a simple screen
Window 0,0,0,320,100,0,"Click in me to close",0,1
ev.l=WaitEvent                                ;wait for an event.
```

See Also:

Event, GadgetHit, MenuHit, ItemHit, SubHit, EventWindow

Function: **Event**

Statement: **Event**

Modes: Amiga

Description:

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 occured, **Event** will return 0.

Example:

```
;
; key press idcmp example program
;
Screen 0,3                ;open a simple screen
ScreensBitMap 0,0         ;pick up it's bitmap
DefaultIDCMP $400          ;set 'key press' IDCMP for window

Window 0,0,0,320,200,$1000,"Press a key to exit",0,1

While Event=0            ;while no event...
 Circlef Rnd(300),Rnd(200),Rnd(100),Rnd(8)
Wend
```

See Also:

WaitEvent

Function: **EventWindow**

Syntax: **EventWindow**

Modes: Amiga/Blitz

Description:

EventWindow may be used to determine in which window the most recent window event occured. Window events are detected by use of either the **WaitEvent** or **Event** commands.

EventWindow return the window object number in which the most recent window event occured.

Example:

```
;
; EventWindow exmaple program   NOTE: hit 'Esc' to exit this example!
;
Screen 0,3                        ;open a screen and 4 windows
Window 0,0,0,160,100,$100f,"Window 0",1,2
Window 1,160,0,160,100,$100f,"Window 1",1,2
Window 2,0,100,160,100,$100f,"Window 2",1,2
Window 3,160,100,160,100,$100f,"Window 3",1,2

Repeat
 ev.l=WaitEvent                   ;wait for an event
 Use Window lw                    ;use LAST event window
  InnerCls                        ;cls inside area of window
  Use Window EventWindow          ;use THIS event window
  WLocate 0,0                     ;text cursor to top left...
  Print "Event here!"             ;tell 'em about it
  lw=EventWindow                  ;make THIS window LAST window
Until Inkey$=Chr$(27)             ;escape to quit!
```

See Also:

WaitEvent, Event

Statements: **FlushEvents**

Syntax: **FlushEvents** *[IDCMP_Flag]*

Modes: Amiga/Blitz

Description:

When window events occur in Blitz 2, 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 occured between executions of **WaitEvent** or **Event** will be stored in a queue for later use. However, there may be situations where you want to ignore this backlog of events. This is what **FlushEvents** is for.

Executing **FlushEvents** with no parameters will completely clear Blitz 2's internal event queue, leaving you with no outstanding events. Supplyng an *IDCMP_Flag* parameter will only clear events of the specified type from the event queue.

See Also:

WaitEvent, Event

Function: **GadgetHit**

Syntax: GadgetHit

Modes: Amiga

Description:

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 posess the same identification numbers, you may also need to use **EventWindow** to tell exactly which gadget was hit.

Example:

```
;
; simple gadget list example program using gadget hit
;
Screen 0,3                               ;simple Intuition screen
TextGadget 0,20,20,0,1,"Click here"      ;make up a gadgetlist...
TextGadget 0,20,40,0,2,"Or in here"      ;...
TextGadget 0,20,60,0,3,"Quit here"       ;...

Window 0,0,0,320,200,0,"Window and gadgets",0,1,0

Repeat
 Repeat
  ev.l=WaitEvent                 ;wait for an event.
 Until ev=$40                    ;but only 'gadget released'
 If GadgetHit=3 Then End         ;if gadget was #3, then end
Forever
```

See Also:

WaitEvent, Event

Function: **MenuHit**

Syntax: **MenuHit**

Modes: Amiga

Description:

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 the 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.

Example:

```
;
; simple menu example program
;
Screen 0,3                   ;open a simple Intuition screen

Window 0,0,0,320,200,0,"Window with menus",0,1

MenuColour 2                 ;change menu rendering pens
MenuTitle 0,0,"Menus"        ;create a simple menu
MenuItem 0,0,0,0,"Item"      ;with only one item in it.
MenuItem 0,0,0,1,"Quit"      ;and a quit item!
SetMenu 0                    ;add it to window
While MenuHit<>0
 ev.l=WaitEvent
Wend
```

See Also:

WaitEvent, Event, ItemHit, SubHit

Function: **ItemHit**

Syntax: **ItemHit**

Modes: Amiga

Description:

ItemHit returns the identification number of the menu item that caused the last menu event.

Example:

```
;
; exit on quit menu program example
;
Screen 0,3                   ;open a simple screen

Window 0,0,0,320,200,0,"Window with menus",0,1

MenuColour 2                 ;change menu drawing pen
MenuTitle 0,0,"Menus"        ;title of menu 0
MenuItem 0,0,0,0,"First"     ;item 0...
MenuItem 0,0,0,1,"Second"    ;item 1...
MenuItem 0,0,0,2,"Third"     ;item 2...
MenuItem 0,0,0,3,"Quit"      ;item 3...
SetMenu 0                    ;attach menulist to window

Repeat
 WaitEvent
Until ItemHit=3              ;quit when 'Quit' selected.
```

See Also:

WaitEvent, Event, MenuHit, SubHit

Function: **SubHit**

Syntax: **SubHit**

Modes: Amiga

Description:

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.

Example:

```
;
; subitems program example
;
Screen 0,3                          ;open a simple screen

Window 0,0,0,320,200,0,"Window with menus",0,1

MenuColour 2                        ;set menu drawing pens
MenuTitle 0,0,"Menus"               ;menu title...
MenuItem 0,0,0,0,"More "+Chr$(187)  ;item 0.
SubItem 0,0,0,0,0,"Quit"            ;sub item 0
SetMenu 0                           ;attach menulist

Repeat
 WaitEvent
Until SubHit=0
```

See Also:

WaitEvent, Event, MenuHit, ItemHit

Function: **MButtons**

Syntax: **MButtons**

Modes: Amiga

Description:

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**.

The following are the values returned for the buttons by **MButtons**.

Button	Down	Up
Left	1	5
Right	2	6

Example:

```
;
; mbuttons program example
;
Screen 0,3                ;open a simple Intuition window

Window 0,0,0,320,200,$1000,"Click right button to exit",0,1

Repeat
 WaitEvent
Until MButtons=6
```

See Also:

WaitEvent, Event

Function : **RawKey**

Syntax: **RawKey**

Modes: Amiga

Description:

RawKey returns the raw key code of a key that caused the most recent 'key press' event.

Example:

```
;
; qualifiers and keyboard events example
;
Screen 0,3

Window 0,0,0,320,200,0,"Type a control character to quit",0,1

While (Qualifier AND $8) = 0
 ev=WaitEvent
 WLocate 0,0
 a$=Inkey$
 Print Hex$(RawKey)
Wend
```

See Also:

WaitEvent, Event, Qualifier, Inkey$

Function: **Qualifier**

Syntax: **Qualifier**

Modes: Amiga

Description:

Qualifier will return 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.

See Also:

WaitEvent, Event, RawKey, Inkey$

Statement: **WPlot**

Syntax: **WPlot** *X,Y,Colour*

Modes: Amiga

Description:

WPlot plots a pixel in the currently used window at the coordinates *X,Y* in the colour specified by *Colour.*

Example:

```
;
; wplot example
;
Screen 0,3
Window 0,0,0,320,200,0,"",0,1

For t=1 To 40
 For g=1 To 40
  WPlot t,g,2
 Next
Next

MouseWait
```

Statement: **WBox**

Syntax: **WBox** *X1,Y1,X2,Y2,Colour*

Modes: Amiga

Description:

WBox 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*.

Example:

```
;
; wbox example program
;
Screen 0,3
Window 0,0,0,320,200,0,"Boxes",0,1

For t=1 To 1000
 WBox Rnd(320),Rnd(200),Rnd(300),Rnd(200),Rnd(8)
Next

MouseWait
```

Statement: **WCircle**

Syntax: **WCircle** *X,Y,Radius,Colour*

Modes: Amiga

Description:

WCircle allows you to draw a circle in the 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.

Example:

```
;
; wcircle example program
;
Screen 0,3
Window 0,0,0,320,200,0,"Circles",0,1

For t=1 To 1000
 WCircle Rnd(320),Rnd(200),Rnd(300),Rnd(8)
Next

MouseWait
```

Statement: WEllipse

Syntax: **WEllipse** *X,Y,X Radius,Y Radius,Colour*

Modes: Amiga

Description:

WEllipse draws an ellipse in the currently used window. You specify the centre of the ellipse with the coordinates *X,Y*. *X Radius* specifies the horizontal radius of the ellipse, *Y Radius* the vertical radius.

Colour refers to the colour in which to draw the ellipse.

Example:

```
;
; wellipse example program
;
Screen 0,3
Window 0,0,0,320,200,0,"Ellipses",0,1

For t=1 To 1000
  WEllipse Rnd(320),Rnd(200),Rnd(300),Rnd(300),Rnd(8)
Next

MouseWait
```

Statement: WLine

Syntax: **WLine** *X1,Y1,X2,Y2[,Xn,Yn..],Colour*

Modes: Amiga

Description:

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 to be drawn.

Example:

```
;
; wline example program
;
Screen 0,3
Window 0,0,0,320,200,0,"A Polygon",0,1
Wline 150,10,200,60,150,110,100,60,160,10,3
MouseWait
End
```

Statement: **WCls**

Syntax: **WCls** *[Colour]*

Modes: Amiga

Description:

WCls will clear the currently used window to clour 0, or *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..

Example:

```
;
; wcls example
;
Screen 0,3
Window 0,0,0,320,200,$400,"Window Cls",0,1
WCls 2
MouseWait
```

See Also:

InnerCls

Statement: **InnerCls**

Syntax: **InnerCls** *[Colour]*

Modes: Amiga

Description:

InnerCls will clear only the inner portion of the currently used window. It will not clear the titlebar or borders as **Cls** would do if your window was not opened with the GIMMEZEROZERO flag set. If *colour* is specfied, then that colour will be used to clear the window.

Example:

```
;
; innercls example
;
Screen 0,3
Window 0,0,0,320,200,0,"Not a GIMMEZEROZERO window",0,1
InnerCls 2
MouseWait
```

See Also:

WCls

Statement: **WScroll**

Syntax: **WScroll** *X1,Y1,X2,Y2,Delta X,Delta Y*

Modes: Amiga

Description:

WScroll will 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.

Statement: **Cursor**

Syntax: **Cursor** *Thickness*

Modes: Amiga

Description:

Cursor will 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.

Example:

```
;
; cursor example
;
Screen 0,3                                    ;open a simple creen
Window 0,0,0,320,200,0,"Cursor types",0,1     ;and a window
Print "This is a block cursor."               ;show a block cursor
a$=Edit$("Hello",10)
Cursor 1                                ;change cursor to underline
Print "This is an underline one."
a$=Edit$("Hello",10)
End
```

Function: **Editat**

Syntax: **Editat**

Modes: Amiga

Description:

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.

Example:

```
;
; cursor example with edit$
;
Screen 0,3                                      ;open a simple creen
Window 0,0,0,320,200,0,"Cursor types",0,1       ;and a window
Print "This is a block cursor."                 ;show a block cursor
a$=Edit$("Hello",10)
Cursor 1                                        ;change cursor to underline
Print "This is an underline one."
a$=Edit$("Hello",10)
End
```

See Also:

EditFrom, Edit$, Edit

Statement: **EditFrom**

Syntax: **EditFrom** *[Characterpos]*

Modes: Amiga

Description:

EditFrom allows you to control how the **Edit$** and **Edit** functions operate when used within windows.

If a *Characterpos* parameter is specified, then the next time an edit function is executed, editting will commence at the specified character position (0 being the first character position).

Also, editting may be terminated not just by the use of the 'return' key, but also by any non printable character (for example, '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 *Characterpos* is omitted, **Edit$** and **Edit** return to normal - editting always beginning at character postition 0, and 'return' being the only way to exit.

Example:

```
;
;a simple full screen editor.
;
Dim lines$(20)                    ;enough for 20 lines
Screen 0,0,0,320,172,2,0,"Blitz Edit - Hit 'ESC' to Quit",1,2
Window 0,0,0,320,172,$1900,"",2,1
y=1                               ;starting line
WLocate 0,12                      ;prepare to number lines
Format "##"

For k=1 To 20                     ;loop to print line numbers.
  NPrint k,":"
Next
```

```
Repeat

 Repeat  ;first, we should handle all events (gadgets, menus etc)
  ev.l=Event
  Select ev ;this is where actual handling should take place.

  End Select
 Until ev=0              ;until no more events to handle

 WLocate 24,y*8+4          ;now, prepare to edit 'current' line
 EditFrom x             ;start at character position 'x'
 lines$(y)=Edit$(lines$(y),37)
 x=Editat                  ;character position at time of 'edit exit'

 Select EditExit          ;How did they exit?
  Case 13                      ;Return?
   x=0                    ;back to left of line
   If y<20 Then y+1       ;and possibly down a line
  Case 28                      ;Up arrow?
   If y>1 Then y-1 ;possibly up a line
  Case 29                      ;Down arrow?
   If y<20 Then y+1       ;possibly down a line
 End Select

Until EditExit=27              ;until 'Escape' hit
```

See Also:

Editat, EditExit, Edit$, Edit

Function: **EditExit**

Syntax: **EditExit**

Modes: Amiga/Blitz

Description:

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.

Example:

```
;
; edit exit example
;
Screen 0,2                ;open a simple screen
Window 0,0,0,320,200,$1000,"Press ESCAPE to quit",0,1
Repeat
 FlushEvents          ;to get rid of outstanding window events.
 WLocate 0,0           ;to top left...
 EditFrom Editat       ;edit from last quit position
 a$=Edit$(a$,38)
```

```
Until EditExit=27
```

See Also:

EditFrom, Editat, Edit$, Edit

Statement: **WindowFont**

Syntax: **WindowFont** *IntuiFont#*

Modes: Amiga

Description:

WindowFont sets the font for the currently used window. Any further printing to this window will be in the specified font. *IntuiFont#* specifies a previously initialized intuifont object created using **LoadFont**.

Example:

```
;
; window font example
;
Screen 0,3                              ;a simple screen and window...
Window 0,0,0,320,200,$1000,"Groovy font",0,1
LoadFont 0,"topaz.font",11                ;get into topaz 11
WindowFont 0                           ;set this as the font for the window
Print "This is in Topaz 11"              ;show the font
MouseWait
End
```

See Also:

LoadFont

Statement: **WColour**

Syntax: **WColour** *Foreground Colour[,Background Colour]*

Modes: Amiga

Description:

WColour 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.

Example:

```
;
; wcolour example
;
Screen 0,3                              ;open Intuition screen and window..
Window 0,0,0,320,200,$1000,"Colours",0,1
```

```
For T=1 To 7              ;foreground colour loop
 For G=1 To 7              ;background colour loop
  WColour T,G               ;set window colour
  Print "Wow! "           ;print some text...
 Next
 NPrint ""
Next

MouseWait
End
```

See Also:

WJam

Statement: **WJam**

Syntax: **WJam** *Jammode*

Modes: Amiga

Description:

WJam sets the text drawing mode of the currently used window. These drawing modes allow you to do inverted, complemented and other types of gaphics. The drawing modes can be OR'ed together to create a combination of them. Here are the different modes.

Mode	Value	Description
Jam1	0	This draws only the foreground colour and leaves the background transparent. Eg For the letter O, any empty space (inside and outside the letter) will be transparent.
Jam2	1	This draws both the foreground and background to the window. Eg With the letter O again, the O will be drawn, but any clear area (inside and outside) will be drawn in the current background colour.
Complement	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.
Inversvid	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.

Example:

```
;
; wjam examples
;
Screen 0,3                        ;open Intuition screen and window..
Window 0,0,0,320,200,0,"DrawModes",0,1
Print "OverLapping characters"  ;print some stuff in different modes
WJam 0
```

```
Print "Hello"
WLocate 0,0
Print "Bye"
WJam 1
Print "Overwriting characters"
Print "Hello"
WLocate 0,16
Print "Bye"
Print "Bye"
Print "Complemented characters disappear"
WJam 2
Print "Hello"
WLocate 0,32
Print "Hello"
WJam 4
Print "This is in inverse video"
MouseWait
End
```

See Also:

WColour

Statement: **Activate**

Syntax: **Activate** *Window#*

Modes: Amiga

Description:

Activate will active the window specified by *Window#*.

Example:

```
;
; activate windows example
;
Screen 0,2
Window 0,0,0,320,100,0,"Window 1",0,1
Window 1,0,100,320,100,0,"Window 2",0,1
Activate 0
Print "Hello"
Activate 1
Print "Good Bye"
MouseWait
End
```

Statement: **Menus**

Syntax: **Menus** *On*| *Off*

Modes: Amiga

Description:

The **Menus** command 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.

Statement: **WPointer**

Syntax: **WPointer** *Shape#*

Modes: Amiga

Description:

WPointer allows you to determine the mouse pointer imagery used in the currently used window. *Shape#* specifies an initialized shape object the pointer is to take it's appearance from, and must be of 2 bitplanes depth (4 colours).

Example:

```
;
; wpointer example
;
Screen 0,2                         ;Open a simple screen and window.
Window 0,0,0,320,200,$1000,"New Pointer",0,1
LoadShape 0,"TestPointer"          ;load a shape.
WPointer 0                         ;make it the pointer
MouseWait
```

Statement: **WMove**

Syntax: **WMove** *X,Y*

Modes: Amiga

Description:

WMove will move the current window to a screen position specified by *X* and *Y*.

Example:

```
;
; wmove example
;
Screen 0,2
Window 0,0,0,100,100,$1000,"Moving window!",0,1

For k=1 To 50
```

```
 WMove k,k
Next

MouseWait
```

See Also:

WSize

Statement: **WSize**

Syntax: **WSize** *Width,Height*

Modes: Amiga

Description:

WSize will alter the width and height of the current window to the values specified by *Width* and *Height*.

Example:

```
;
; wsize example
;
Screen 0,2
Window 0,0,0,10,10,$1000,"",0,1
VWait 100
WSize 320,100
Print "Click Mouse to Quit"
MouseWait
```

See Also:

WMove

Function: **WMouseX**

Syntax: WMouseX

Modes: Amiga

Description:

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.

Example:

```
;
; wmousex and wmousey example
;
Screen 0,2
Window 0,0,0,320,200,0,"Window",0,1

While Joyb(0)=0
 WLocate 0,0
 Print WMouseX," ",WMouseY
Wend
```

See Also:

WMouseY

Function: **WMouseY**

Syntax: **WMouseY**

Modes: Amiga

Description:

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.

See Also:

WMouseX

Function: **EMouseX**

Syntax: **EMouseX**

Modes: Amiga/Blitz

Description:

EMouseX will return the horizontal position of the mouse pointer at the time the most recent window event occured. Window events are detected using the **WaitEvent** or **Event** commands.

Example:

```
;
; emousex & y program example
;
Screen 0,3
ScreensBitMap 0,0
;
```

```
Repeat                      ;repeat...
 ev.l=WaitEvent               ;wait for a window event
 If MButtons=1              ;if left mouse button down...
  x=EMouseX:y=EMouseY         ;grab mouse x and y at time of event
  Repeat                  ;repeat...
   ev2.l=WaitEvent          ;wait for a window event
   If ev2=$10                ;mouse moved?
    Wline x,y,EMouseX,EMouseY,1     ;join up a line...
    x=EMouseX:y=EMouseY    ;grab new mouse x and y
   EndIf
  Until MButtons=5          ;until left button up
 EndIf
Until ev=$200               ;until window closed.
```

See Also:

EMouseY, WMouseX, WMouseY, WaitEvent, Event

Function: **EMouseY**

Syntax: **EMouseY**

Modes: Amiga/Blitz

Description:

EMouseY will return the vertical position of the mouse pointer at the time the most recent window event occured. Window events are detected using the **WaitEvent** or **Event** commands.

See Also:

EMouseX, WMouseX, WMouseY, WaitEvent, Event

Function: **WCursX**

Syntax: **WCursX**

Modes: Amiga

Description:

WCursX returns the horizontal location of the text cursor of the currently used window. The text cursor position may be set using **WLocate.**

Example:

```
;
; wcursx example
;
Screen 0,2
Window 0,0,0,320,200,0,"Window",0,1
For T=1 To 5
```

```
  Print WCursX;" ";
Next
MouseWait
End
```

See Also:

WCursY, WLocate

Function: **WCursY**

Syntax: **WCursY**

Modes: Amiga

Description:

WCursY returns the vertical location of the text cursor of the currently used window. The text cursor position may be set using **WLocate**.

Example:

```
;
; wcursy example
;
Screen 0,2
Window 0,0,0,320,200,0,"Window",0,1
For T=1 To 5
  NPrint WCursY
Next
MouseWait
End
```

See Also:

WCursX, WLocate

Statement: **WLocate**

Syntax: **WLocate** *X,Y*

Modes: Amiga/Blitz

Description:

WLocate is 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 it's own text cursor position, therefore changing the text cursor position of one window will not affect any other window's text cursor position.

See Also:

WCursx, WCursy

Function: **WindowX**

Syntax: **WindowX**

Modes: Amiga

Description:

WindowX returns the horizontal pixel location of the top left corner of the currently used window, relative to the screen the window appears in.

Example:

```
;
; windowx example
;
Screen 0,2
Window 0,10,0,300,200,0,"Window",0,1
Print WindowX
MouseWait
End
```

See Also:

WindowY, WindowWidth, WindowHeight

Function: **WindowY**

Syntax: **WindowY**

Modes: Amiga

Description:

WindowY returns the vertical pixel location of the top left corner of the currently used window, relative to the screen the window appears in.

Example:

```
;
; windowy example program
;
Screen 0,2
Window 0,0,10,320,180,0,"Window",0,1
Print WindowY
MouseWait
```

See Also:

WindowX, WindowWidth, WindowHeight

Function: **WindowWidth**

Syntax: **WindowWidth**

Modes: Amiga

Description:

WindowWidth returns the pixel width of the currently used window.

Example:

```
Screen 0,2
Window 0,0,0,320,200,0,"WindowWidth",0,1
Print WindowWidth
MouseWait
End
```

See Also:

WindowX, WindowY, WindowHeight

Statement: **WindowHeight**

Syntax: **WindowHeight**

Modes: Amiga

Description:

WindowHeight returns the pixel height of the currently used window.

See Also:

WindowX, WindowY, WindowWidth

Function: **InnerWidth**

Syntax: **InnerWidth**

Modes: Amiga

Description:

InnerWidth returns the pixel width of the area inside the border of the currently used window.

See Also:

InnerHeight

Function: **InnerHeight**

Syntax: **InnerHeight**

Modes: Amiga

Description:

InnerHeight returns the pixel height of the area inside the border of the currently used window.

See Also:

InnerWidth

Function: **WTopOff**

Syntax: **WTopOff**

Modes: Amiga

Description:

WTopOff returns the number of pixels between the top of the current window border and the inside of the window.

See Also:

WLeftOff

Function: **WLeftOff**

Syntax: **WLeftOff**

Modes: Amiga

Description:

WLeftOff returns the number of pixels between the left edge of the current window border and the inside of the window.

Statement: **SizeLimits**

Syntax: **SizeLimits** *Min Width,Min Height,Max Width,Max Height*

Modes: Amiga

Description:

SizeLimits 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.

Example:

```
;
; sizelimits program example
;
Screen 0,2                        ;A simple screen
SizeLimits 20,20,150,150          ;set limits for windows
Window 0,0,0,100,100,15,"SizeLimits",0,1
Print "Click RMB"
Print "to quit"
While Joyb(0)<>2
Wend
```

Function: **RastPort**

Syntax: **RastPort** (*Winodw#*)

Modes: Amiga

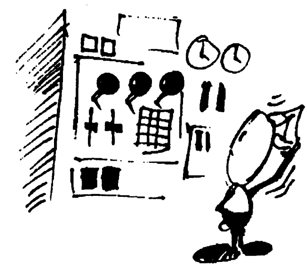
Description:

RastPort returns the specified Window's RastPort address. Many commands in the graphics.library and the like require a RastPort as a parameter.

See Also:

ViewPort

26. Gadgets



Blitz 2 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.

The following is a table of the gadget flags and the gadget types which they are relevant to:

Bit #	Meaning	Text	String	Prop	Shape
0	Toggle On/Off	yes	no	no	yes
1	Relative to Right Side of Window	yes	yes	yes	yes
2	Relative to Bottom of Window	yes	yes	yes	yes
3	Size Relative to Width of Window	no	no	yes	no
4	Size Relative to Height of Window	no	no	yes	no
5	Box Select	yes	yes	yes	yes
6	Prop Gadget has Horizontal Movement	no	no	yes	no
7	Prop Gadget Has Vertical Movement	no	no	yes	no
8	No Border around Prop Gadget Container	no	no	yes	no

Note:

If Relative Right is set the gadgets 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 witdh as WindowWidth+GadgetWidth as it does height when relative size flags are set.

Statement: **TextGadget**

Syntax: **TextGadget** *GadgetList#,X,Y,Flags,Id,Text$*

Modes: Amiga/Blitz

Description:

The **TextGadget** 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.

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 gadget has been selected. *Id* may be any positive, non-zero number.

Text$ is the actual text you want the gadget to contain.

Example:

```
;
; textgadget example
;
TextGadget 0,8,180,0,1," EXIT "              ;add to gadgetlist 0
TextGadget 0,216,180,0,2," STAY HERE "  ;add this too
Screen 0,3                                      ;open screen
Window 0,0,0,320,200,$100f,"GADGETS!",1,2,0
Repeat                                          ;wait for 'EXIT'
Until WaitEvent=64 AND GadgetHit=1
```

See Also:

ShapeGadget, StringGadget, PropGadget

Statement: **GadgetPens**

Syntax: **GadgetPens** *Foreground Colour[,Background Colour]*

Modes: Amiga/Blitz

Description:

GadgetPens 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.

Example:

```
;
; gadget pens example program
;
BorderPens 3,3      ;change gadget border colours
```

```
TextGadget 0,8,DispHeight-16,0,1," OK "
GadgetPens 2          ;change gadget pens
TextGadget 0,320-88,DispHeight-16,0,2," CANCEL "
;
Screen 0,3            ;open a screen
RGB 1,0,15,0          ;set some colours
RGB 2,15,0,0
RGB 3,15,15,15
;
Window 0,0,0,320,DispHeight,$100f,"My Window",0,0,0
;
Repeat                ;wait for gadget hit...
Until WaitEvent=64
```

See Also:

GadgetJam

Statement: **GadgetJam**

Syntax: **GadgetJam** *Jammode*

Modes: Amiga/Blitz

Description:

GadgetJam 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.

See Also:

GadgetPens

Statement: **ShapeGadget**

Syntax: **ShapeGadget** *GadgetList#,X,Y,Flags,Id,Shape#*

Modes: Amiga/Blitz

Description:

The **ShapeGadget** command allows you to create gadgets with graphic imagery. The *Shape#* parameter refers to a shape object containing the graphics you want the gadget to contain.

All aother parameters are identical to those in **TextGadget**.

Example:

```
;
; shapegadget example
;
```

```
Screen 0,3
ScreensBitMap 0,0

For k=7 To 1 Step -1
 Circlef 16,16,k*2,k
Next
GetaShape 0,0,0,32,32

ShapeGadget 0,148,50,0,1,0
TextGadget 0,140,180,0,2," EXIT "
Window 0,0,0,320,200,$100f,"More Gadgets!",1,2,0

Repeat
Until WaitEvent=64 AND GadgetHit=2
```

See Also:

TextGadget, StringGadget, PropGadget

Statement: **Toggle**

Syntax: **Toggle** *GadgetList#,Id,On| Off*

Modes: Amiga/Blitz

Description:

Toggle allows you to 'turn on' or 'turn off' a text or shape gadget created with a 'toggle' flags setting.

Toggle will not affect the gadget's imagery if it is already displayed.

See Also:

TextGadget

Statement: **StringGadget**

Syntax: **StringGadget** *GadgetList#,X,Y,Flags,Id,Maxlen,Width*

Modes: Amiga/Blitz

Description:

StringGadget allows you 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 from 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.

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 gadget has been selected. *Id* may be any positive, non-zero number.

Maxlen refers to the maximum number of characters which may appear in this gadgets.

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.

Example:

```
;
; string gadget example
;
StringGadget 0,80,16,0,1,40,160  ;add string gadget to gadgetlist 0
StringGadget 0,80,32,0,2,40,160  ;add another string gadget
TextGadget 0,8,180,0,3," EXIT "  ;add an 'EXIT' gadget

Screen 0,3                       ;open a screen, and window...

Window 0,0,0,320,200,$100f,"String Gadgets!",1,2,0

WLocate 8,8                      ;print some text...
Print "Name:"
WLocate 8,24                     ;and some more...
Print "Address:"

Repeat                           ;wait for 'QUIT'
Until WaitEvent=64 AND GadgetHit=3
```

See Also:

TextGadget, ShapeGadget, PropGadget, StringText, ActivateString, ClearString, ResetString

Function: **StringText$**

Syntax: **StringText$** *(GadgetList#,Id)*

Modes: Amiga/Blitz

Description:

The **Stringtext$** function allows you to determine the current contents of a string gadget. **StringText$** will return a string of characters representing the string gadgets contents.

Example:

```
;
; activated string gadget example
;
StringGadget 0,128,16,0,1,40,160   ;make a string gadget
TextGadget 0,8,180,0,2," EXIT "    ;and an exit gadget
Screen 0,3                         ;open screen and window
```

```
Window 0,0,0,320,200,$100f,"StringText$ demo...",1,2,0

WLocate 4,8
Print "Type your name:"
ActivateString 0,1              ;turn on string gadget

Repeat                          ;wait for 'EXIT'
 a.l=WaitEvent
 If a=64 AND GadgetHit=1        ;string entry complete?
   WLocate 8,96
   Print Centre$("Hello there "+StringText$(0,1),38)
   ClearString 0,1
   Redraw 0,1
   ActivateString 0,1
 EndIf
Until a=64 AND GadgetHit=2
```

See Also:

StringGadget

Statement: **ActivateString**

Syntax: **ActivateString** *Window#,Id*

Modes: Amiga/Blitz

Description:

ActivateString 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 for the mouse before the keyboard.

Example:

```
;
; string gadget input example
;
StringGadget 0,128,16,0,1,40,160     ;make a string gadget
TextGadget 0,8,180,0,2," EXIT "      ;and an exit gadget
Screen 0,3                           ;open screen and window

Window 0,0,0,320,200,$100f,"String Gadget Activated...",1,2,0

WLocate 4,8                     ;prompt...
Print "Type your name:"
ActivateString 0,1              ;turn on string gadget
Repeat                          ;wait for 'EXIT'
Until WaitEvent=64 AND GadgetHit=2
```

See Also:

StringGadget, ResetString, ClearString

Statement: **ResetString**

Syntax: **ResetString** *GadgetList#,Id*

Modes: Amiga/Blitz

Description:

ResetString allows you to 'reset' a string gadget. This will cause the string gadget's cursor position to be set to the leftmost position.

Example:

```
;
; reset string gadget example
;
StringGadget 0,128,16,0,1,40,160      ;make a string gadget
TextGadget 0,8,180,0,2," EXIT "       ;and an 'exit' gadget
Screen 0,3                            ;open a screen and a window...

Window 0,0,0,320,200,$100f,"ResetString demo...",1,2,0

WLocate 4,8
Print "Type your name:"               ;prompt...
ActivateString 0,1                    ;click on string gadget for them...

Repeat                                ;do...
 a.l=WaitEvent                        ;wait for something to happen
 If a=64 AND GadgetHit=1              ;string entry complete?
  ResetString 0,1                     ;yes, reset string gadget...
  ActivateString 0,1                  ;and re-activate it!
 EndIf
Until a=64 AND GadgetHit=2            ;until 'QUIT' hit.
```

See Also:

StringGadget, ActivateString, ClearString

Statement: **ClearString**

Syntax: **ClearString** *GadgetList#,Id*

Modes: Amiga/Blitz

Description:

ClearString may be used to clear, or erase, the text in the specified string gadget. The cursor position will also be moved to the leftmost 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.

Example:

```
;
; clear string gadget example
;
StringGadget 0,128,16,0,1,40,160       ;make a string gadget
TextGadget 0,8,180,0,2," EXIT " ;and an 'EXIT' gadget
Screen 0,3                             ;open intuition screen and window...
Window 0,0,0,320,200,$100f,"ClearString demo...",1,2,0
WLocate 4,8
Print "Type your name:"       ;prompt...
ActivateString 0,1              ;actiavte string gadget
Repeat                          ;do...
    a.l=WaitEvent                 ;wait for something to happen!
    If a=64 AND GadgetHit=1        ;string entry done?
        ClearString 0,1        ;yup - clear text...
        Redraw 0,1                 ;re draw gadget...
        ActivateString 0,1          ;and re-activate string gadget
    EndIf
Until a=64 AND GadgetHit=2
```

See Also:

StringGadget, **ActivateString**, **ResetString**

Statement: **SetString**

Syntax: **SetString** *GadgetList#,ID,String$*

Modes: Amiga/Blitz

Description:

SetString may be used to initialize 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 exeucte **ReDraw** to display the change.

See also:

StringGadget, **GadgetText$**, **ReDraw**

Statement: **PropGadget**

Syntax: **PropGadget** *GadgetList#,X,Y,Flags,Id,Width,Height*

Modes: Amiga/Blitz

Description:

The **PropGadget** command is 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.

Proportional gadgets are commonly used for the 'R G B' 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 '.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 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 gadgets position, relative to the 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.

Id is a unique, non zero number which allows you to identify when the gadget is manipulated.

Proportional gadgets may be altered using the **SetVProp** and **SetHProp** commands, and read using the **VPropPot**, **VPropBody**, **HPropPot** and **HPropBody** functions.

Example:

```
;
; propgadget example
;
PropGadget 0,8,16,5,1,8,64          ;add 'Red' slider to gadgetlist 0
PropGadget 0,24,16,5,2,8,64          ;add 'green' slider
PropGadget 0,40,16,5,3,8,64          ;add 'red' slider
TextGadget 0,8,180,0,4," QUIT "      ;and, of course, a 'QUIT' button.

For k=1 To 3                  ;go through sliders...
 SetVProp 0,k,0,1/16             ;set them all to pot=0, body=1/16
Next

Screen 0,3                    ;an intuition screen
RGB 0,0,0,0                   ;colour 0 to black (same as sliders)

Window 0,0,0,320,200,$100f,"R G B Sliders!",1,3,0

WLocate 4,72                   ;label sliders...
Print "R G B"

Repeat                        ;do...
 a.l=WaitEvent                   ;wait for something to happen.
 Select a                     ;what happened?
 Case 32 ;gadget down                ;a gadget was pressed...
  If GadgetHit<>4                ;if it wasn't quit...
   Repeat                     ;do...
    RGB 0,VPropPot(0,1)*16,VPropPot(0,2)*16,VPropPot(0,3)*16
   Until Event=64                ;until slider released
```

```
  EndIf
 Case 64                          ;a gadget was released...
  If GadgetHit=4 Then End          ;if it was 'QUIT', then do so..
  RGB 0,VPropPot(0,1)*16,VPropPot(0,2)*16,VPropPot(0,3)*16
 End Select
Forever
MouseWait
```

See Also:

SetHProp, SetVProp, HPropPot, HPropBody, VPropPot, VPropBody

Statement: **SetHProp**

Syntax: **SetHProp** *GadgetList#,Id,Pot,Body*

Modes: Amiga/Blitz

Description:

SetHProp is used to alter the horizontal slider qualities of a proportional gadget. Both *Pot* and *Body* should be in the range 0 through 1.

If **SetHProp** is executed while the specified gadget is already displayed, execution of the **ReDraw** command will be necessary to display the changes.

For a full discussion on proportional gadgets, please refer to the **PropGadget** command.

See Also:

SetVPropPot, HPropPot, HPropBody, VPropPot, VPropBody

Statement: **SetVProp**

Syntax: **SetVProp** *GadgetList#,Id,Pot,Body*

Modes: Amiga/Blitz

Description:

SetVProp is used to alter the vertical slider qualities of a proportional gadget. Both *Pot* and *Body* should be in the range 0 through 1.

If **SetVProp** is executed while the specified gadget is already displayed, execution of the **ReDraw** command will be necessary to display the changes.

For a full discussion on proportional gadgets, please refer to the **PropGadget** command.

See Also:

SetHPropPot, HPropPot, HPropBody, VPropPot, VPropBody

Function: **HPropPot**

Syntax: **HPropPot** *(GadgetList#,Id)*

Modes: Amiga/Blitz

Description:

The **HPropPot** function allows you to determine the current 'pot' setting of a proportional gadget. **HPropPot** will return a number from 0 up to, but not including, 1, reflecting the gadgets current horizontal pot setting.

Please refer to the **PropGadget** command for a full discussion on proportional gadgets.

See Also:

VPropPot, HPropBody, VPropBody

Function: **HPropBody**

Syntax: **HPropBody** *(GadgetList#,Id)*

Modes: Amiga/Blitz

Description:

The **HPropBody** function allows you to determine the current 'body' setting of a proportional gadget. **HPropBody** will return a number from 0 up to, but not including, 1, reflecting the gadgets current horizontal body setting.

Please refer to the **PropGadget** command for a full discussion on proportional gadgets.

See Also:

VPropPot, HPropPot, VPropBody

Function: **VPropPot**

Syntax: **VPropPot** *(GadgetList#,Id)*

Modes: Amiga/Blitz

Description:

The **VPropPot** function allows you to determine the current 'pot' setting of a proportional gadget. **VPropPot** will return a number from 0 up to, but not including, 1, reflecting the gadgets current vertical pot setting.

Please refer to the **PropGadget** command for a full discussion on proportional gadgets.

See Also:

HPropPot, HPropBody, VPropBody

Function: **VPropBody**

Syntax: **VPropBody** *(GadgetList#,Id)*

Modes: Amiga/Blitz

Description:

The **VPropBody** function allows you to determine the current 'body' setting of a proportional gadget.

VPropBody will return a number from 0 up to, but not including, 1, reflecting the gadgets current vertical body setting.

Please refer to the **PropGadget** command for a full discussion on proportional gadgets.

See Also:

VPropPot, **HPropPot**, **HPropBody**

Statement: **ReDraw**

Syntax: **ReDraw** *Window#,id*

Modes: Amiga/Blitz

Description:

ReDraw 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.

Statement: **Borders**

Syntax: **Borders** *[On| Off]| [Width,Height]*

Modes: Amiga/Blitz

Description:

Borders serves 2 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 it's border, *Width* referring to the left/right spacing, and *Height* to the above/below spacing.

Example:

```
;
;
; gadget borders example
;
```

```
Borders Off                          ;turn borders off...
TextGadget 0,8,16,0,1,"NO BORDERS"   ;add a gadget
Borders On                           ;turn borders on...
TextGadget 0,8,32,0,2,"BORDERS"      ;add a gadget
Borders 16,8                         ;set border spacing...
TextGadget 0,8,64,0,3,"BIG BORDERS!" ;add a gadget
Borders 8,4                          ;this is default border spacing
TextGadget 0,8,180,0,4," QUIT "      ;add 'QUIT' gadget
Screen 0,3                           ;open screen, and window...
Window 0,0,0,320,200,$100f,"Select a gadget...",1,2,0
Repeat                               ;wait for 'QUIT'
Until WaitEvent=64 AND GadgetHit=4
```

Statement: **BorderPens**

Syntax: **BorderPens** *Highlight Colour,Shadow Colour*

Modes: Amiga/Blitz

Description:

BorderPens allows you to control the colours used when gadget borders are created. Gadget borders may be created by the **TextGadget, StringGadget** and **GadgetBorder** commands.

HighLight Colour refers to the colour of the top and left edges of the border, while *Shadow Colour* refers to the right and bottom edges.

The default value for *HighLight Colour* is 1. The default value for *Shadow Colour* is 2.

Example:

```
;
; borderpens example program
;
BorderPens 2,1  ;change gadget border colours
TextGadget 0,8,DispHeight-16,0,1," OK "
TextGadget 0,320-88,DispHeight-16,0,2," CANCEL "
;
Screen 0,2        ;open a screen
RGB 0,6,6,6       ;set some colours
RGB 1,15,15,15
RGB 2,0,0,0
RGB 3,15,15,0
;
Window 0,0,0,320,DispHeight,$100f,"My Window",0,0,0
;
Repeat            ;wait for gadget hit...
Until WaitEvent=64
```

See Also:

Borders

Statement: **GadgetBorder**

Syntax: **GadgetBorder** *X,Y,Width,Height*

Modes: Amiga/Blitz

Description:

The **GadgetBorder** command may be used to draw a rectangular border into the currently used window.

Proportional gadgets and shape gadgets do not have borders automatically created for them. The **GadgetBorder** command may be used, once a window is opened, to render borders around these gadgets.

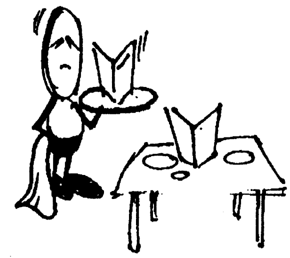
X,Y, *Width* and *Height* refer to the position of the gadget a border is required around. **GadgetBorder** will automatically insert sapces between the gadget and the border. The **Borders** command may be used to alter the amount of spacing.

Of course, **GadgetBorder** may be used to draw a border around any arbitary area, regardless of whether or not that area contains a gadget.

See Also:

Borders

27. Menus



Blitz 2 supports many commands for the creation and use of Intuition menus.

Menus are created through the use of MenuList objects. Each menulist contains an entire set of menu titles, menu items and possibly sub menu items.

Menulists are attached to windows through the **SetMenu** command.

Each window may use a separate menulist, allowing you to attach relevant menus to different windows.

Statement: MenuTitle

Syntax: **MenuTitle** *Menulist#,Menu,Title$*

Modes: Amiga/Blitz

Description:

MenuTitle is 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 leftmost menu. Menutitles 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.

Example:

```
;
;simple menus example
;
MenuTitle 0,0,"PROJECT" ;create a menu title
MenuItem 0,0,0,0,"QUIT" ;and an item...
MenuTitle 0,1,"EDIT"   ;create another menu title
MenuItem 0,0,1,0,"CUT"  ;and give it some items...
MenuItem 0,0,1,1,"COPY" ;...

Screen 0,3,"Menus Example"      ;an intuition screen, and below, a window

Window 0,0,12,320,DispHeight-12,$100f,"Hold Down the right mouse button...",0,1

SetMenu 0                       ;attach menulist to currently used window

Repeat                          ;wait...until 'QUIT' selected.
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=0
```

See Also:

MenuItem, ShapeItem, SubItem, ShapeSub

Statement: MenuItem

Syntax: **MenuItem** *MenuList#,Flags,Menu,Item,Itemtext$[,Shortcut$]*

Modes: Amiga/Blitz

Description:

MenuItem is used to create a **text** menu item. Menu items appear vertically below menu titles when the mouse is moved over a menu title with the right mouse button held down.

Flags affects the operation of the menu item.

A value of 0 creates a stand 'seleect' 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 be in 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 the topmost item. Menu items should be added in 'top down' order, with menu 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.

Example:

```
;
; toggle items in menu example
;
MenuTitle 0,0,"Testing"              ;create a menu title
MenuItem 0,0,0,0,"Load    ","L"      ;and an item (with shortcut!)
MenuItem 0,0,0,1,"Save","S"          ;another item...
MenuItem 0,1,0,2,"  ASCII ?"         ;this is a toggle item!
MenuItem 0,0,0,3,"QUIT!!!!!"

Screen 0,3                 ;an intuition screen

Window 0,0,12,320,DispHeight-12,$100f,"Select a Menu...",1,2

SetMenu 0
```

```
Repeat                              ;wait for 'QUIT'...

;check for certain menus here...

Until WaitEvent=256 AND MenuHit=0 AND ItemHit=3
```

See Also:

MenuTitle, ShapeItem, SubItem, ShapeSub

Statement: ShapeItem

Syntax: **ShapeItem** *MenuList#,Flags,Menu,Item,Shape#*

Modes: Amiga/Blitz

Description:

ShapeItem is used to create a graphical menu item.

Shape# refers to a previously initialized shape object to be used as the menu item's graphics.

All other parameters are identical to those for **MenuItem**.

Example:

```
;
; shapeitem example
;
Screen 0,3                    ;open an intuition screen
ScreensBitMap 0,0                 ;borrow it's bitmap
BitMapOutput 0              ;send 'Print' to the bitmap
Cls                         ;clear bitmap
Print "LoadSaveQuit"          ;write some text
GetaShape 0,0,0,32,8           ;get 'Load' as shape 0
GetaShape 1,32,0,32,8          ;get 'Save' as shape 1
GetaShape 2,64,0,32,8          ;get 'Quit' as shape 2
Cls                         ;clear bitmap again
MenuTitle 0,0,"PROJECT" ;make a menu title

For k=0 To 2                  ;process all 3 shapes
 Scale k,4,2                  ;stretch 'em a bit
 ShapeItem 0,0,0,k,k                ;use shape as a menu item
Next

Window 0,0,0,320,DispHeight,$100f,"Select a menu!",1,2
SetMenu 0                      ;attach menulist to window

Repeat                        ;wait for 'QUIT'
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=2
```

See Also:

MenuTitle, MenuItem, SubItem, ShapeSub

Statement: **SubItem**

Syntax: **SubItem** *MenuList#,Flags,Menu,Item,Subitem,Subitemtext$[,Shortcut$]*

Modes: Amiga/Blitz

Description:

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 **SubItem** command.

Item specifies the menu item to attach the sub item to.

Subitem refers to the number of the sub menu item 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$* specifies the actual text for the sub item. As with menu items, sub items may have an optional keyboard shortcut character, specified using the *Shortcut$ paramater.

All other parameters are identical to the **MenuItem** command.

Example:

```
;
; subitems menu example
;
MenuTitle 0,0,"PROJECT"              ;make a menu title
MenuItem 0,0,0,0,"LOAD "+Chr$(187)     ;item...
SubItem 0,0,0,0,0,"PICTURE"           ;sub items...
SubItem 0,0,0,0,1,"BRUSH"             ;
MenuItem 0,0,0,1,"QUIT"
Screen 0,3                           ;open a screen and window

Window 0,0,12,320,DispHeight-12,$100f,"Select a menu...",1,2
SetMenu 0    ;attach menu list

Repeat                              ;wait for 'QUIT'
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=1
```

See Also:

MenuTitle, MenuItem, ShapeItem, ShapeSub

Statement: **ShapeSub**

Syntax: **ShapeSub** *MenuList#,Flags,Menu,Item,Subitem,Shape#*

Modes: Amiga/Blitz

Description:

ShapeSub allows you to create a graphic sub menu item. *Shape#* specifies a previously created shape object to be used as the sub item's grpahics.

All other parameters are identical to those in **SubItem**.

Statement: **SetMenu**

Syntax: **SetMenu** *MenuList#*

Modes: Amiga/Blitz

Description:

SetMenu is used to attach a menulist to the currently used window. Each window may have only one menulist attached to it.

Statement: **MenuGap**

Syntax: **MenuGap** *X Gap,Y Gap*

Modes: Amiga/Blitz

Description:

Executing **MenuGap** before creating any menu titles, items or sub items, allows you to control the layout of the menu.
X Gap refers to an amount, specified in pixles, to be inserted to the left and right of all menu items and sub menu items. *Y Gap* refers to an amount, again in pixels, to be inserted above and below all menu items and sub menu items.

Example:

```
;
; menugap example
;
MenuGap 32,16              ;set a BIG gap
MenuTitle 0,0,"PROJECT" ;set up MenuList 0...
MenuItem 0,0,0,0,"LOAD"
MenuItem 0,0,0,1,"SAVE"
MenuItem 0,0,0,2,"QUIT"
MenuTitle 0,1,"EDIT"
MenuItem 0,0,1,0,"CUT"
MenuItem 0,0,1,1,"COPY"
MenuItem 0,0,1,2,"PASTE"
Screen 0,3                  ;open an intuition screen and window...
Window 0,0,0,320,DispHeight,$100f,"Select a menu...",1,2
SetMenu 0                   ;attach menulist
Repeat                      ;wait for 'QUIT'
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=2
```

Statement: **SubItemOff**

Syntax: **SubItemOff** *X Offset,Y Offset*

Modes: Amiga/Blitz

Description:

SubItemOff allows you to control the relative position of the top of a list of sub menu items, in relation to their associated menu item.

Whenver 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.

Example:

```
;
; subitemoff example
;
MenuTitle 0,0,"Test"
MenuItem 0,0,0,0,"More "+Chr$(187)+" "
SubItemOff 60,8
SubItem 0,0,0,0,0,"One Sub Menu Item..."
SubItem 0,0,0,0,1,"Two Sub Menu Items.."
MenuItem 0,0,0,1,"QUIT"
Screen 0,3
Window 0,0,0,320,DispHeight,$100f,"Select a menu...",1,2
SetMenu 0
Repeat
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=1
```

Statement: **MenuState**

Syntax: **MenuState** *MenuList#[,Menu[,Item[,Subitem]]],On| Off*

Modes: Amiga/Blitz

Description:

The **MenuState** command allows you to turn menus, or sections of menus, on or off.

MenuState with just the *MenuList#* parameter may be 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.

Statement: **MenuColour**

Syntax: **MenuColour** *Colour*

Modes: Amiga/Blitz

Description:

MenuColour allows you to determine what colour any menu item or sub item text is rendered in. **MenuColour** should be executed before the approprate menu item commands.

Example:

```
;
;menucolour example
;
MenuTitle 0,0,"COLOUR"        ;set up menu title
MenuColour 1                  ;next item made will be in colour 1...
MenuItem 0,0,0,0,"LOAD" ;this is it
MenuColour 2                  ;now colour 2
MenuItem 0,0,0,1,"SAVE" ;
MenuColour 3                  ;and 3...
MenuItem 0,0,0,2,"QUIT"
Screen 0,3                    ;open an intuition screen and window
Window 0,0,0,320,DispHeight,$100f,"Select a menu...",1,2
SetMenu 0                     ;attach our menus
Repeat                        ;wait for 'QUIT'
Until WaitEvent=256 AND MenuHit=0 AND ItemHit=2
```

Function: **MenuChecked**

Syntax: **MenuChekced** *(MenuList#,Menu,Item[,Subitem])*

Modes: Amiga/Blitz

Description:

The **MenuChecked** function 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, **MenuChecked** will return 'true' (-1). If not, **MenuChecked** will return 'false' (0).

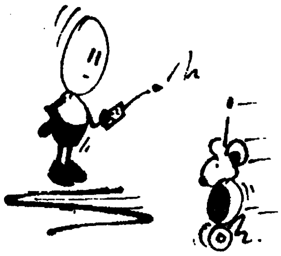
Example:

```
;
; enable checking on menus example using menuchecked
;
MenuTitle 0,0,"TEST!"                 ;create menu title
MenuItem 0,1,0,0," OK TO QUIT?"       ;a toggle menu item
MenuItem 0,0,0,1,"QUIT"               ;an ordinary one.
Screen 0,3                            ;open screen and window...
Window 0,0,0,320,DispHeight,$140f,"Select a menu...",1,2

Repeat
 a.l=WaitEvent          ;wait for somthing to happen
 If a=256 AND ItemHit=1          ;is it 'QUIT' ?
  If MenuChecked(0,0,0) ;is item 0 'on' (checked)?
   End             ;Yup - go ahead and quit
  Else
   WLocate 0,0;else, tell user
   Print "Quit Not Enabled!"
  EndIf
 EndIf
Forever
```

See Also:

MenuItem, ShapeItem, SubItem, ShapeSub

28. BRexx



The Blitz 2 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 your programs 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.

Statement: **AbsMouse**

Syntax: **AbsMouse** *X,Y*

Modes: Amiga

Description:

AbsMouse 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.

Example:

```
;
; brex absmouse program example
;
AbsMouse 0,0        ; This will move the mouse pointer to
                    ; the upper left of the display

AbsMouse 319,199    ;This will approximately 'centre' the mouse
                    ;pointer on the display

AbsMouse 639,399    ;This will move the mouse pointer to the lower
                    ;right of the display
MouseWait
```

See Also:

RelMouse

Statement: **RelMouse**

Syntax: **RelMouse** *X Offset, Y Offset*

Modes: Amiga

Description:

RelMouse allows you to move the mouse pointer a relative distance from it's current location. Positive *offset* parameters will move the pointer rightwards and downwards, while negative *offset* parameters will move the pointer leftwards and upwards.

Example:

```
;
; brex relmouse program example
;
AbsMouse 0,0                ;move pointer to upper left

For k=1 To 100
 RelMouse 1,1              ;across and down 100 times
Next

MouseWait
```

See Also:

AbsMouse

Statement: **MouseButton**

Syntax: **MouseButton** *Button,On/ Off*

Modes: Amiga

Description:

MouseButton 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*).

Example:

```
;
; brex mousebutton program example
;
low=DispHeight*2-1    ;allow for NTSC or PAL
AbsMouse 639,low       ;Move mouse pointer to lower right.
MouseButton 0,On      ;Click down left button.
AbsMouse 319,low/2     ;move mouse pointer to middle
MouseButton 0,Off     ;Release left button.
MouseWait
```

See Also:

ClickButton

Statement: **ClickButton**

Syntax: **ClickButton** *Button*

Modes: Amiga

Description:

ClickButton is 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.

Example:

```
;
; brex clickbutton program example
;
TextGadget 0,32,32,0,1," CLICK ME "
Screen 0,3
Window 0,0,0,320,200,$100f,"Magic!",1,2,0
AbsMouse 40,0
For k=1 To 18
RelMouse 4,4
Next
ClickButton 0
MouseWait
```

Statement: **Type**

Syntax: **Type** *String$*

Modes: Amiga

Description:

Type 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.

Example:

```
;
; brex recording program example
;
Type "Hello There!"
MouseWait
```

Statement: **Record**

Syntax: **Record** *[Tape#]*

Modes: Amiga

Description:

Record 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.

Example:

```
;
; brex recording program example
;
Type "Hello There!"
MouseWait

NPrint "Play with the mouse, then hit the right mouse button."
AbsMouse 0,0
Record 0                ;begin recording.
While Joyb(0)<>2
Wend
Record                  ;finish recording
AbsMouse 0,0
PlayBack 0
MouseWait
```

See Also:

PlayBack, TapeTrap

Statement: **PlayBack**

Syntax: **PlayBack** *[Tape#]*

Modes: Amiga

Description:

PlayBack 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.

Example:

```
;
; brex program example
;
low=DispHeight*2-1      ;allow for NTSC or PAL displays
TapeTrap 0               ;start creating a tape
QuietTrap On              ;set recording mode to quiet.
AbsMouse 639,low
MouseButton 0,On
AbsMouse 639,low/2
MouseButton 0,Off
TapeTrap                ;Turn off trapping.
PlayBack 0               ;Play it Back!
MouseWait
```

See Also:

Record, TapeTrap, QuickPlay

Statement: **QuickPlay**

Syntax: **QuickPlay** *On*/ *Off*

Modes: Amiga

Description:

QuickPlay 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 it's 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.

See Also:

PlayBack

Statement: **PlayWait**

Syntax: **PlayWait**

Modes: Amiga

Description:

PlayWait may be used to halt program flow until a **PlayBack** of a tape has finished.

See Also:

PlayBack

Function: **XStatus**

Syntax: **XStatus**

Modes: Amiga

Description:

XStatus returns a value depending upon the current state of the BRexx system. Possible return values and their meanings are as follows:

Value:	**Meaning:**
0	BRexx is currently inactive. No tapes are either 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.

See Also:

Record, TapeTrap, PlayBack

Statement: **SaveTape**

Syntax: **SaveTape** *Tape#,Filename$*

Modes: Amiga

Description:

SaveTape allows you to save a previously created tape object out to disk. This tape may later be reloaded using **LoadTape**.

See Also:

LoadTape

Statement: **LoadTape**

Syntax: **LoadTape** *Tape#,Filename$*

Modes: Amiga

Description:

LoadTape allows you to load a tape object previously saved with **SaveTape** for use with the **PlayBack** command.

See Also:

SaveTape

Statement: **TapeTrap**

Syntax: **TapeTrap** *[Tape#]*

Modes: Amiga

Description:

TapeTrap 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.

See Also:

Record, PlayBack, QuietTrap

Statement: **QuietTrap**

Syntax: **QuietTrap** *On| Off*

Modes: Amiga

Description:

QuietTrap determines the way in which any **TapeTrap**ping 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.

See Also:

TapeTrap

Statement: **MacroKey**

Syntax: **MacroKey** *Tape#,Rawkey,Qualifier*

Modes: Amiga

Description:

MacroKey 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.

Example:

```
;
; brex macrokey program example
;
TapeTrap 0
QuietTrap On
AbsMouse 0,0
AbsMouse 639,0
AbsMouse 639,399
AbsMouse 0,399
AbsMouse 0,0
TapeTrap
MacroKey 0,128,0
NPrint "Hit F1..."
MouseWait
```

Statement: **FreeMacroKey**

Syntax: **MacroKey** *Rawkey,Qualifier*

Modes: Amiga

Description:

FreeMacroKey causes a previously defined macro key to be removed so that a BRex tape is no longer attatched to it.

See Also:

MacroKey

The Blitz 2 Objects

The following chapter covers the Blitz 2 objects. Objects are structures such as bitplanes and shapes that Blitz dynamically allocates and controls.

The information included in the listing at the end of this chapter can be used to 'intimately' manipulate Blitz 2 objects.

Firstly the adress of the structure in memory needs to be found. The following is an example of picking up the address of the bitplane data from a shape:

```
INCLUDE "blitz2incs.bb" ;or use the resident file!

LoadShape 0,"myshape"

*a.shape=Addr Shape(0)  ;a is a pointer type to a shape type

d.l=*a\_data            ;the long variable d now holds the shapes image location
```

Modules are sound-tracker compatible files used to sequence music.

```
NEWTYPE.module
_mt_data.l      ;00: NULL if no module present,
                ;    else pointer to module data
_length.l       ;04: length of module data
                ;08: sizeof
End NEWTYPE
```

BlitzFonts are any 8x8 fonts able to be used to print in Blitz mode.

```
NEWTYPE.blitzfont
_font.l         ;00: NULL if no font present,
                ;    else pointer to GFX TextFont struct
                ;04: sizeof
End NEWTYPE
```

Screens are simply pointers to Intuition screens.

```
NEWTYPE.screen
_screen.l       ;00: NULL if no screen present,
                ;    else pointer to INTUITION screen struct
                ;04: sizeof
End NEWTYPE
```

Menus are simply pointers to a list of Intuition menus.

```
NEWTYPE.menulist
_menu.l          ;00: NULL if no menu present,
               ;  else pointer to linked INTUTIION
               ;  menu items
               ;04: sizeof
End NEWTYPE
```

IntuiFonts are normal Amiga fonts used with windows and screens.

```
NEWTYPE.intuifont
_fontname.l      ;00: Pointer to name of font
_ysize.w         ;04: height of font
_pad.w           ;06:
_font.l          ;08: NULL if no font present,
               ;  else pointer to GFX TextFont struct
_pad2.b(4)       ;12:
               ;16: sizeof
End NEWTYPE
```

Shapes are used for all the blitting commands.

```
NEWTYPE.shape
_pixwidth.w      ;00: NULL if no shape present,
               ;  else pixel width of shape
_pixheight.w     ;02: pixel height of shape
_depth.w         ;04: depth, in bitplanes, of shape
_ebwidth.w       ;06: even byte width of shape
_bltsize.w       ;08: BLTSIZE of shape
_xhandle.w       ;10: horizontal handle of shape
_yhandle.w       ;12: vertical handle of shape
_data.l          ;14: pointer to graphic data - Plane1, Plane2...
_cookie.l        ;18: pointer to one bitplane cookiecut
_onebpmem.w      ;22: memory taken by one bitplane of shape
_onebpmemx.w     ;24: memory taken by one bitplane of shape,
               ;  plus an extra word per bitplane per
               ;  vertical pixel
_allbpmem.w      ;26: memory taken by entire shape.
_allbpmemx.w     ;28: memory taken by entire shape, plus an
               ;  extra word per bitplane per vertical
               ;  pixel
_pad.b(2)        ;30:
               ;32: sizeof
End NEWTYPE
```

Tapes are used by BRexx for recording a series of events that can 'drive' Intuition.

```
NEWTYPE.tape
_ielist.l        ;00: NULL if no tape present,
               ;  else pointer to list of InputEvents
_timevalhi.l     ;04: high 4 bytes of timeval of first event
_timevallo.l     ;08: low 4 bytes of timeval of first event
_pad.b(4)        ;12:
               ;16: sizeof
End NEWTYPE
```

Stencils are used for Blits that need to go behind some things and infront of others

```
NEWTYPE.stencil
_ebwidth.w        ;00: NULL if no stencil present,
            ;  else even byte width
_height.w         ;02: height of stencil
_data.l           ;04: pointer to one bitplane of stencil data
            ;08: sizeof
End NEWTYPE
```

A queue item holds information for the **UnQueue** command.

```
NEWTYPE.queueitem
_mod.w            ;00: blitter BLTDMOD value
_bltsize.w        ;02: blitter BLTSIZE value
_depth.w          ;04: depth, in bitplanes, of bitmap
_bitmap.l         ;06: bitmap object QBLIT was made to
_offset.l         ;10: offset into bitmap QBLIT was made at
End NEWTYPE
```

Queues are like list headers that point to a series of queue items.

```
NEWTYPE.queue
*_current.queueitem ;00: pointer to where to add next QBLIT
            ;  QueueItem
*_first.queueitem  ;04: NULL if no Queue present,
            ;  else pointer to start of
            ;  .QueueItem block
_length.l         ;08: Length of allocated queue memory
_pad.b(4)         ;12:
            ;16: sizeof
End NEWTYPE
```

FieldItems are used for Random Access files.

```
NEWTYPE.fielditem
*_next.fielditem   ;00: For linked list.
_data.l           ;04: pointer to where data comes from
            ;  or goes to
_lenth.l          ;08: length of above data
End NEWTYPE
```

The file structure is used to control open DOS files in Blitz 2.

```
NEWTYPE.file
_handle.l         ;00: NULL if no file present,
            ;  else dos file handle of file
_reclen.l         ;04: Byte length of 'Fields' for this file
_pad.b(4)         ;08:
*_fields.fielditem ;12: list of field items
_buffer.l         ;16: buffer for my own read/write routines
_bufflen.w        ;20: length of above buffer
_flags.w          ;22: =0 : buffer not altered,
            ;  <0 : buffer written to,
            ;  >0 : seek necessary when buffer flushed
_valid.w          ;24: number of valid bytes in buffer
_seekoff.w        ;26: seek (position) offset into buffer
```

```
_seek.l          ;28: dos seek of start of buffer
End NEWTYPE
```

The Palette structure is used to hold sets of colours for both Screens and Slices

```
NEWTYPE.palette
_numcols.w        ;00: NULL if no palette present,
               ;   else number of colours (0-31) in palette
_colours.w(32)    ;02: Max of 32 RGB words.
               ;
_lowcol.w         ;66: low colour for cycling,
               ;   <0 = end of cycling table.
_hicol.w          ;68: high colour for cycling
_speed.w          ;70: speed of cycling - 16384 = max.
               ;   if speed = 0, then cycle downwards,
               ;   else cycle upwards.
_var.w            ;72: variable to add speed to.
               ;
               ;   (More possible cycling entries)
               ;
               ;128: sizeof
End NEWTYPE
```

Buffers are used by the BBlit command to hold background information that a BBlit overwrites.

```
NEWTYPE.buffer
_current.l        ;00: Pointer to current point in buffer
               ;   to add BBLIT info to.
_first.l          ;04: NULL if no buffer present,
               ;   else pointer to beginning of buffer
               ;   memory.
_length.l         ;08: length in bytes of buffer memory.
_pad.b(4)         ;12:
               ;16: sizeof
End NEWTYPE
```

A gadgetlist simply points to a list of Intuition gadgets.

```
NEWTYPE.gadgetlist
_gadgets.l        ;00: NULL if no gadgetlist present,
               ;   else pointer to first gadget
               ;   in list of Intuition gadgets.
               ;04: sizeof
End NEWTYPE
```

Window objects hold information about the Intuition window they point to.

```
NEWTYPE.window
_window.l         ;00: NULL if no window present,
               ;   else pointer to Intuition
               ;   window struct
_cursx.w          ;04: horizontal cursor position in window
_cursy.w          ;06: vertical cursor position in window
_pointer.l        ;08: pointer optional window pointer
               ;   sprite data.
_length.l         ;12: length of window pointer sprite data.
```

```
        ;16: sizeof
End NEWTYPE
```

Slices hold information concerning the copper lists used to create Blitz mode displays.

```
NEWTYPE.slice
_ypos.w          ;00: NULL if no slice present,
            ;    else vertical position of slice
_flags.w         ;02: slice flags
_numbitplanes.w  ;04: number of bitplanes available in slice
_numsprites.w    ;06: number of sprites available in slice
_numcolours.w    ;08: number of colours available in slice
_bitplanes.l     ;10: pointer to address, in copper list,
            ;    of bitplane MOVEs
_sprites.l       ;14: pointer to address, in copper list,
            ;    of sprite MOVEs
_colours.l       ;18: pointer to address, in copper list,
            ;    of colour MOVEs
_BPLCON1.l       ;22: pointer to address, in copper list,
            ;    of word MOVEd to BPLCON1
_BPLCON2.l       ;26: pointer to address, in copper list,
            ;    of word MOVEd to BPLCON2
_pad.b(2)        ;30:
            ;32: sizeof
End NEWTYPE
```

BitMaps hold pointers and other information.

```
NEWTYPE.bitmap
_ebwidth.w       ;00: even byte width of bitmap
_height.w        ;02: pixel height of bitmap
_depth.w         ;04: depth, in bitplanes, of bitmap
_pad.b(2)        ;06:
_data.l(8)       ;08: Max of 8 pointers to bitplanes
_pad2.b(22)      ;40:
_isreal.w        ;62: =0 : no bitmap present
            ;    <0 : bitmap present
            ;    >0 : bitmap present, but not ours
            ;64: sizeof
End NEWTYPE
```

Sound objects hold information concerning the noisier commands in Blitz 2.

```
NEWTYPE.sound
_data.l          ;00: NULL if no sound present,
            ;    else pointer to sound data
_period.w        ;04: period of sound
_length.w        ;06: length, in words, of sound data
_loop.l          ;08: repeat to loop position of sound
_looplength.w    ;12: length of looping section, in words
_pad.b(2)        ;14:
            ;16: sizeof
End NEWTYPE
```

Sprite objects contain the information required by the Blitz 2 sprite library.

```
NEWTYPE.sprite
_data.l         ;00: NULL if no sprite present,
              ;  else pointer to sprite data
_height.w       ;04: height of sprite, in pixels, plus
              ;  an extra 1
_channels.w     ;06: number of sprite channels required
              ;  to display sprite
_flags.w        ;08: low byte = pix width of sprite,
              ;  hi bit = 1 if 16 colour sprite
_nextoff.w      ;10: difference, in bytes, between seperate
              ;  sprites for separate sprite channels
_xhandle.w      ;12: horizontal handle for sprite
_yhandle.w      ;14: vertical handle for sprite
              ;16: sizeof
End NEWTYPE
```

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.

The errors are grouped under the following headers:

General Syntax Errors

Procedure Related Errors

Constants Related Errors

Expression Evaluation Errors

Illegal Errors

Library Based Errors

Include Errors

Program Flow Based Errors

Type Based Errors

Conditional Compiling Errors

Resident Based Errors

Macro Based Errors

Array Errors

Interrupt Based Errors

Label Errors

Direct Mode Errors

Select ... End Select Errors

Blitz Mode / Amiga Mode Errors

Strange Beast Errors

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 paramaters. Use the HELP key for correct number and meaning of parameters with Blitz][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 syntatically incorrect.

Illegal End Procedure

The statement or function end is syntatically 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 only defined from the primary 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 syntatically 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

The 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.

Illgeal Absolute

The Absolute location specified must be defined and in range.

Illegal Displacement

The 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'

Blitz][cannot find the library routine referred to by a Token, usually caused by the library not being included in DefLibs 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

The 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

Blitz 2 has a limit of 32K for any blocks of code such as IF..ENDIF blocks.

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 to Long

Blitz 2 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 types

The Exchange command can only swap two variables of the same type.

Can't Exchange NewTypes

The Exchange command can not 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

Blitz 2 cannot find the Resident file listed in the Options requester. Check the pathname.

Macro Based Errors

Macro Buffer Overflow

The Options requester in the Blitz 2 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 wellas somewhere 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 Blitz 2. 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, Blitz 2 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 paramater with the Dim statement.

Array not yet Dim'd

See Array not found.

Array not Dim'd

See Array not found.

Interupt Based Errors

End SetInt without SetInt

SetInt..SetInt is a block directive and both commands must be present.

SetInt without End SetInt

SetInt..SetInt is a block directive and both commands must be present.

Can't use Set/ClrInt 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 Set/ClrInt in Local Mode

Error handling can only be defined by the primary code.

End SetInt without SetInt

SetInt..SetInt is a block directive and both commands must be present.

SetInt without End SetInt

SetInt..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 the same name.

Illegal Label Name

Refer to the Programming in Blitz][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

The library is only available from within your code.

Illegal direct mode command

Direct mode is unable to execute the 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 / Amiga 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

Optimizer 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.

Amiga Library Routines

BLITZLIBS:AMIGALIBS currently supports the EXEC, DOS, GRAPHICS, INTUITION and DISKFONT amiga libraries.

Parameter details for each command are given in brackets and are also available via the Blitz 2 keyboard help system.

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.

The relative offsets from the library base and 68000 register parameters are included for the convenience of the assembler programmer.

When using library calls an underscore character (_) should follow the token name.

An asterisk (*) preceding routine names specifies that the calls are private and should not be called from Blitz 2.

EXEC

-30 Supervisor(userFunction)(a5)

---- special patchable hooks to internal exec activity ---

-36 *execPrivate1()()
-42 *execPrivate2()()
-48 *execPrivate3()()
-54 *execPrivate4()()
-60 *execPrivate5()()
-66 *execPrivate6()()

--- module creation ---

-72 InitCode(startClass,version)(d0/d1)
-78 InitStruct(initTable,memory,size)(a1/a2,d0)
-84 MakeLibrary(funcInit,structInit,libInit,dataSize,segList)(a0/a1/a2,d0/d1)
-90 MakeFunctions(target,functionArray,funcDispBase)(a0/a1/a2)
-96 FindResident(name)(a1)
-102 InitResident(resident,segList)(a1,d1)

--- diagnostics ---

-108 Alert(alertNum)(d7)
-114 Debug(flags)(d0)

--- interrupts ---

-120 Disable()()
-126 Enable()()
-132 Forbid()()
-138 Permit()()
-144 SetSR(newSR,mask)(d0/d1)
-150 SuperState()()
-156 UserState(sysStack)(d0)
-162 SetIntVector(intNumber,interrupt)(d0/a1)
-168 AddIntServer(intNumber,interrupt)(d0/a1)
-174 RemIntServer(intNumber,interrupt)(d0/a1)
-180 Cause(interrupt)(a1)

--- memory allocation ---

-186 Allocate(freeList,byteSize)(a0,d0)
-192 Deallocate(freeList,memoryBlock,byteSize)(a0/a1,d0)
-198 AllocMem(byteSize,requirements)(d0/d1)
-204 AllocAbs(byteSize,location)(d0/a1)
-210 FreeMem(memoryBlock,byteSize)(a1,d0)
-216 AvailMem(requirements)(d1)
-222 AllocEntry(entry)(a0)
-228 FreeEntry(entry)(a0)

--- lists ---

-234 Insert(list,node,pred)(a0/a1/a2)
-240 AddHead(list,node)(a0/a1)
-246 AddTail(list,node)(a0/a1)
-252 Remove(node)(a1)
-258 RemHead(list)(a0)
-264 RemTail(list)(a0)
-270 Enqueue(list,node)(a0/a1)
-276 FindName(list,name)(a0/a1)

--- tasks ---

-282 AddTask(task,initPC,finalPC)(a1/a2/a3)
-288 RemTask(task)(a1)
-294 FindTask(name)(a1)
-300 SetTaskPri(task,priority)(a1,d0)
-306 SetSignal(newSignals,signalSet)(d0/d1)
-312 SetExcept(newSignals,signalSet)(d0/d1)
-318 Wait(signalSet)(d0)
-324 Signal(task,signalSet)(a1,d0)
-330 AllocSignal(signalNum)(d0)
-336 FreeSignal(signalNum)(d0)
-342 AllocTrap(trapNum)(d0)
-348 FreeTrap(trapNum)(d0)

--- messages ---

-354 AddPort(port)(a1)
-360 RemPort(port)(a1)
-366 PutMsg(port,message)(a0/a1)
-372 GetMsg(port)(a0)
-378 ReplyMsg(message)(a1)
-384 WaitPort(port)(a0)
-390 FindPort(name)(a1)

--- libraries ---

-396 AddLibrary(library)(a1)
-402 RemLibrary(library)(a1)
-408 OldOpenLibrary(libName)(a1)
-414 CloseLibrary(library)(a1)
-420 SetFunction(library,funcOffset,newFunction)(a1,a0,d0)
-426 SumLibrary(library)(a1)

--- devices ---

-432 AddDevice(device)(a1)
-438 RemDevice(device)(a1)
-444 OpenDevice(devName,unit,ioRequest,flags)(a0,d0/a1,d1)
-450 CloseDevice(ioRequest)(a1)
-456 DoIO(ioRequest)(a1)
-462 SendIO(ioRequest)(a1)
-468 CheckIO(ioRequest)(a1)
-474 WaitIO(ioRequest)(a1)
-480 AbortIO(ioRequest)(a1)

--- resources ---

-486 AddResource(resource)(a1)
-492 RemResource(resource)(a1)
-498 OpenResource(resName)(a1)

--- private diagnostic support ---

-504 *execPrivate7()()
-510 *execPrivate8()()
-516 *execPrivate9()()

--- misc ---

-522 RawDoFmt(formatString,dataStream,putChProc,putChData)(a0/a1/a2/a3)
-528 GetCC()()
-534 TypeOfMem(address)(a1)
-540 Procure(semaport,bidMsg)(a0/a1)
-546 Vacate(semaport)(a0)
-552 OpenLibrary(libName,version)(a1,d0)

*** functions in Release 1.2 or higher ***

--- signal semaphores (note funny registers found in 1.2 or higher)---

-558 InitSemaphore(sigSem)(a0)
-564 ObtainSemaphore(sigSem)(a0)
-570 ReleaseSemaphore(sigSem)(a0)
-576 AttemptSemaphore(sigSem)(a0)
-582 ObtainSemaphoreList(sigSem)(a0)
-588 ReleaseSemaphoreList(sigSem)(a0)
-594 FindSemaphore(sigSem)(a1)
-600 AddSemaphore(sigSem)(a1)
-606 RemSemaphore(sigSem)(a1)

--- kickmem support ---

-612 SumKickData()()

--- more memory support ---

-618 AddMemList(size,attributes,pri,base,name)(d0/d1/d2/a0/a1)
-624 CopyMem(source,dest,size)(a0/a1,d0)
-630 CopyMemQuick(source,dest,size)(a0/a1,d0)

*** functions in Release 2.0 or higher ***

--- cache ---

-636 CacheClearU()()
-642 CacheClearE(address,length,caches)(a0,d0/d1)
-648 CacheControl(cacheBits,cacheMask)(d0/d1)

--- misc ---

-654 CreateIORequest(port,size)(a0,d0)
-660 DeleteIORequest(iorequest)(a0)
-666 CreateMsgPort()()
-672 DeleteMsgPort(port)(a0)
-678 ObtainSemaphoreShared(sigSem)(a0)

--- even more memory support ---

-684 AllocVec(byteSize,requirements)(d0/d1)
-690 FreeVec(memoryBlock)(a1)
-696 CreatePrivatePool(requirements,puddleSize,puddleThresh)(d0/d1/d2)
-702 DeletePrivatePool(poolHeader)(a0)
-708 AllocPooled(memSize,poolHeader)(d0/a0)
-714 FreePooled(memory,poolHeader)(a1,a0)

--- misc ---

-720 AttemptSemaphoreShared(sigSem)(a0)
-726 ColdReboot()()
-732 StackSwap(newStack)(a0)

--- task trees ---

-738 ChildFree(tid)(d0)
-744 ChildOrphan(tid)(d0)
-750 ChildStatus(tid)(d0)
-756 ChildWait(tid)(d0)

--- future expansion ---

-762 CachePreDMA(address,length,flags)(a0/a1,d1)
-768 CachePostDMA(address,length,flags)(a0/a1,d1)
-774 *execPrivate10()()
-780 *execPrivate11()()
-786 *execPrivate12()()
-792 *execPrivate13()()

DOS

-30 Open(name,accessMode)(d1/d2)
-36 Close(file)(d1)
-42 Read(file,buffer,length)(d1/d2/d3)
-48 Write(file,buffer,length)(d1/d2/d3)
-54 Input()()
-60 Output()()
-66 Seek(file,position,offset)(d1/d2/d3)
-72 DeleteFile(name)(d1)
-78 Rename(oldName,newName)(d1/d2)
-84 Lock(name,type)(d1/d2)
-90 UnLock(lock)(d1)
-96 DupLock(lock)(d1)
-102 Examine(lock,fileInfoBlock)(d1/d2)
-108 ExNext(lock,fileInfoBlock)(d1/d2)
-114 Info(lock,parameterBlock)(d1/d2)
-120 CreateDir(name)(d1)
-126 CurrentDir(lock)(d1)
-132 IoErr()()
-138 CreateProc(name,pri,segList,stackSize)(d1/d2/d3/d4)
-144 Exit(returnCode)(d1)
-150 LoadSeg(name)(d1)
-156 UnLoadSeg(seglist)(d1)
-162 *dosPrivate1()()
-168 *dosPrivate2()()
-174 DeviceProc(name)(d1)
-180 SetComment(name,comment)(d1/d2)
-186 SetProtection(name,protect)(d1/d2)
-192 DateStamp(date)(d1)
-198 Delay(timeout)(d1)
-204 WaitForChar(file,timeout)(d1/d2)
-210 ParentDir(lock)(d1)
-216 IsInteractive(file)(d1)
-222 Execute(string,file,file2)(d1/d2/d3)

*** functions in Release 2.0 or higher ***

---DOS Object creation/deletion---

-228 AllocDosObject(type,tags)(d1/d2)
-234 FreeDosObject(type,ptr)(d1/d2)

---Packet Level routines---

-240 DoPkt(port,action,arg1,arg2,arg3,arg4,arg5)(d1/d2/d3/d4/d5/d6/d7)
-246 SendPkt(dp,port,replyport)(d1/d2/d3)
-252 WaitPkt()()
-258 ReplyPkt(dp,res1,res2)(d1/d2/d3)
-264 AbortPkt(port,pkt)(d1/d2)

---Record Locking---

-270 LockRecord(fh,offset,length,mode,timeout)(d1/d2/d3/d4/d5)
-276 LockRecords(recArray,timeout)(d1/d2)
-282 UnLockRecord(fh,offset,length)(d1/d2/d3)
-288 UnLockRecords(recArray)(d1)

---Buffered File I/O---

-294 SelectInput(fh)(d1)
-300 SelectOutput(fh)(d1)
-306 FGetC(fh)(d1)
-312 FPutC(fh,ch)(d1/d2)
-318 UnGetC(fh,character)(d1/d2)
-324 FRead(fh,block,blocklen,number)(d1/d2/d3/d4)
-330 FWrite(fh,block,blocklen,number)(d1/d2/d3/d4)
-336 FGets(fh,buf,buflen)(d1/d2/d3)
-342 FPuts(fh,str)(d1/d2)
-348 VFWritef(fh,format,argarray)(d1/d2/d3)
-354 VFPrintf(fh,format,argarray)(d1/d2/d3)
-360 Flush(fh)(d1)
-366 SetVBuf(fh,buff,type,size)(d1/d2/d3/d4)

---DOS Object Management---

-372 DupLockFromFH(fh)(d1)
-378 OpenFromLock(lock)(d1)
-384 ParentOfFH(fh)(d1)
-390 ExamineFH(fh,fib)(d1/d2)
-396 SetFileDate(name,date)(d1/d2)
-402 NameFromLock(lock,buffer,len)(d1/d2/d3)
-408 NameFromFH(fh,buffer,len)(d1/d2/d3)
-414 SplitName(name,seperator,buf,oldpos,size)(d1/d2/d3/d4/d5)
-420 SameLock(lock1,lock2)(d1/d2)
-426 SetMode(fh,mode)(d1/d2)
-432 ExAll(lock,buffer,size,data,control)(d1/d2/d3/d4/d5)
-438 ReadLink(port,lock,path,buffer,size)(d1/d2/d3/d4/d5)
-444 MakeLink(name,dest,soft)(d1/d2/d3)
-450 ChangeMode(type,fh,newmode)(d1/d2/d3)
-456 SetFileSize(fh,pos,mode)(d1/d2/d3)

---Error Handling---

-462 SetIoErr(result)(d1)
-468 Fault(code,header,buffer,len)(d1/d2/d3/d4)
-474 PrintFault(code,header)(d1/d2)
-480 ErrorReport(code,type,arg1,device)(d1/d2/d3/d4)
-486 RESERVED

---Process Management---

-492 Cli()()
-498 CreateNewProc(tags)(d1)
-504 RunCommand(seg,stack,paramptr,paramlen)(d1/d2/d3/d4)
-510 GetConsoleTask()()
-516 SetConsoleTask(task)(d1)
-522 GetFileSysTask()()
-528 SetFileSysTask(task)(d1)
-534 GetArgStr()()
-540 SetArgStr(string)(d1)
-546 FindCliProc(num)(d1)
-552 MaxCli()()
-558 SetCurrentDirName(name)(d1)
-564 GetCurrentDirName(buf,len)(d1/d2)
-570 SetProgramName(name)(d1)
-576 GetProgramName(buf,len)(d1/d2)

-582 SetPrompt(name)(d1)
-588 GetPrompt(buf,len)(d1/d2)
-594 SetProgramDir(lock)(d1)
-600 GetProgramDir()()

---Device List Management---

-606 SystemTagList(command,tags)(d1/d2)
-612 AssignLock(name,lock)(d1/d2)
-618 AssignLate(name,path)(d1/d2)
-624 AssignPath(name,path)(d1/d2)
-630 AssignAdd(name,lock)(d1/d2)
-636 RemAssignList(name,lock)(d1/d2)
-642 GetDeviceProc(name,dp)(d1/d2)
-648 FreeDeviceProc(dp)(d1)
-654 LockDosList(flags)(d1)
-660 UnLockDosList(flags)(d1)
-666 AttemptLockDosList(flags)(d1)
-672 RemDosEntry(dlist)(d1)
-678 AddDosEntry(dlist)(d1)
-684 FindDosEntry(dlist,name,flags)(d1/d2/d3)
-690 NextDosEntry(dlist,flags)(d1/d2)
-696 MakeDosEntry(name,type)(d1/d2)
-702 FreeDosEntry(dlist)(d1)
-708 IsFileSystem(name)(d1)

---Handler Interface---

-714 Format(filesystem,volumename,dostype)(d1/d2/d3)
-720 Relabel(drive,newname)(d1/d2)
-726 Inhibit(name,onoff)(d1/d2)
-732 AddBuffers(name,number)(d1/d2)

---Date, Time Routines---

-738 CompareDates(date1,date2)(d1/d2)
-744 DateToStr(datetime)(d1)
-750 StrToDate(datetime)(d1)

---Image Management---

-756 InternalLoadSeg(fh,table,funcarray,stack)(d0/a0/a1/a2)
-762 InternalUnLoadSeg(seglist,freefunc)(d1/a1)
-768 NewLoadSeg(file,tags)(d1/d2)
-774 AddSegment(name,seg,system)(d1/d2/d3)
-780 FindSegment(name,seg,system)(d1/d2/d3)
-786 RemSegment(seg)(d1)

---Command Support----

-792 CheckSignal(mask)(d1)
-798 ReadArgs(template,array,args)(d1/d2/d3)
-804 FindArg(keyword,template)(d1/d2)
-810 ReadItem(name,maxchars,cSource)(d1/d2/d3)
-816 StrToLong(string,value)(d1/d2)
-822 MatchFirst(pat,anchor)(d1/d2)
-828 MatchNext(anchor)(d1)
-834 MatchEnd(anchor)(d1)
-840 ParsePattern(pat,buf,buflen)(d1/d2/d3)

-846 MatchPattern(pat,str)(d1/d2)
-852 * Not currently implemented.
-858 FreeArgs(args)(d1)
-864 *--- (1 function slot reserved here) ---
-870 FilePart(path)(d1)
-876 PathPart(path)(d1)
-882 AddPart(dirname,filename,size)(d1/d2/d3)

---Notification---
-888 StartNotify(notify)(d1)
-894 EndNotify(notify)(d1)

---Environment Variable functions---

-900 SetVar(name,buffer,size,flags)(d1/d2/d3/d4)
-906 GetVar(name,buffer,size,flags)(d1/d2/d3/d4)
-912 DeleteVar(name,flags)(d1/d2)
-918 FindVar(name,type)(d1/d2)
-924 *dosPrivate4()()
-930 CliInitNewcli(dp)(a0)
-936 CliInitRun(dp)(a0)
-942 WriteChars(buf,buflen)(d1/d2)
-948 PutStr(str)(d1)
-954 VPrintf(format,argarray)(d1/d2)
-960 *--- (1 function slot reserved here) ---
-966 ParsePatternNoCase(pat,buf,buflen)(d1/d2/d3)
-972 MatchPatternNoCase(pat,str)(d1/d2)
-978 dosPrivate5()()
-984 SameDevice(lock1,lock2)(d1/d2)

GRAPHICS

-30 BltBitMap
(srcBitMap,xSrc,ySrc,destBitMap,xDest,yDest,xSize,ySize,minterm,mask,tempA)
(a0,d0/d1/a1,d2/d3/d4/d5/d6/d7/a2)

-36 BltTemplate(source,xSrc,srcMod,destRP,xDest,yDest,xSize,ySize)(a0,d0/d1/a1,d2/d3/d4/d5)

--- Text routines ---

-42 ClearEOL(rp)(a1)
-48 ClearScreen(rp)(a1)
-54 TextLength(rp,string,count)(a1,a0,d0)
-60 Text(rp,string,count)(a1,a0,d0)
-66 SetFont(rp,textFont)(a1,a0)
-72 OpenFont(textAttr)(a0)
-78 CloseFont(textFont)(a1)
-84 AskSoftStyle(rp)(a1)
-90 SetSoftStyle(rp,style,enable)(a1,d0/d1)

--- Gels routines ---

-96 AddBob(bob,rp)(a0/a1)
-102 AddVSprite(vSprite,rp)(a0/a1)
-108 DoCollision(rp)(a1)
-114 DrawGList(rp,vp)(a1,a0)

-120 InitGels(head,tail,gelsInfo)(a0/a1/a2)
-126 InitMasks(vSprite)(a0)
-132 RemIBob(bob,rp,vp)(a0/a1/a2)
-138 RemVSprite(vSprite)(a0)
-144 SetCollision(num,routine,gelsInfo)(d0/a0/a1)
-150 SortGList(rp)(a1)
-156 AddAnimOb(anOb,anKey,rp)(a0/a1/a2)
-162 Animate(anKey,rp)(a0/a1)
-168 GetGBuffers(anOb,rp,flag)(a0/a1,d0)
-174 InitGMasks(anOb)(a0)

--- General graphics routines ---

-180 DrawEllipse(rp,xCenter,yCenter,a,b)(a1,d0/d1/d2/d3)
-186 AreaEllipse(rp,xCenter,yCenter,a,b)(a1,d0/d1/d2/d3)
-192 LoadRGB4(vp,colors,count)(a0/a1,d0)
-198 InitRastPort(rp)(a1)
-204 InitVPort(vp)(a0)
-210 MrgCop(view)(a1)
-216 MakeVPort(view,vp)(a0/a1)
-222 LoadView(view)(a1)
-228 WaitBlit()()
-234 SetRast(rp,pen)(a1,d0)
-240 Move(rp,x,y)(a1,d0/d1)
-246 Draw(rp,x,y)(a1,d0/d1)
-252 AreaMove(rp,x,y)(a1,d0/d1)
-258 AreaDraw(rp,x,y)(a1,d0/d1)
-264 AreaEnd(rp)(a1)
-270 WaitTOF()()
-276 QBlit(blit)(a1)
-282 InitArea(areaInfo,vectorBuffer,maxVectors)(a0/a1,d0)
-288 SetRGB4(vp,index,red,green,blue)(a0,d0/d1/d2/d3)
-294 QBSBlit(blit)(a1)
-300 BitClear(memBlock,byteCount,flags)(a1,d0/d1)
-306 RectFill(rp,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3)
-312 BltPattern(rp,mask,xMin,yMin,xMax,yMax,maskBPR)(a1,a0,d0/d1/d2/d3/d4)
-318 ReadPixel(rp,x,y)(a1,d0/d1)
-324 WritePixel(rp,x,y)(a1,d0/d1)
-330 Flood(rp,mode,x,y)(a1,d2,d0/d1)
-336 PolyDraw(rp,count,polyTable)(a1,d0/a0)
-342 SetAPen(rp,pen)(a1,d0)
-348 SetBPen(rp,pen)(a1,d0)
-354 SetDrMd(rp,drawMode)(a1,d0)
-360 InitView(view)(a1)
-366 CBump(copList)(a1)
-372 CMove(copList,destination,data)(a1,d0/d1)
-378 CWait(copList,v,h)(a1,d0/d1)
-384 VBeamPos()()
-390 InitBitMap(bitMap,depth,width,height)(a0,d0/d1/d2)
-396 ScrollRaster(rp,dx,dy,xMin,yMin,xMax,yMax)(a1,d0/d1/d2/d3/d4/d5)
-402 WaitBOVP(vp)(a0)
-408 GetSprite(sprite,num)(a0,d0)
-414 FreeSprite(num)(d0)
-420 ChangeSprite(vp,sprite,newData)(a0/a1/a2)
-426 MoveSprite(vp,sprite,x,y)(a0/a1,d0/d1)
-432 LockLayerRom(layer)(a5)
-438 UnlockLayerRom(layer)(a5)
-444 SyncSBitMap(layer)(a0)
-450 CopySBitMap(layer)(a0)

-456 OwnBlitter()()
-462 DisownBlitter()()
-468 InitTmpRas(tmpRas,buffer,size)(a0/a1,d0)
-474 AskFont(rp,textAttr)(a1,a0)
-480 AddFont(textFont)(a1)
-486 RemFont(textFont)(a1)
-492 AllocRaster(width,height)(d0/d1)
-498 FreeRaster(p,width,height)(a0,d0/d1)
-504 AndRectRegion(region,rectangle)(a0/a1)
-510 OrRectRegion(region,rectangle)(a0/a1)
-516 NewRegion()()
-522 ClearRectRegion(region,rectangle)(a0/a1)
-528 ClearRegion(region)(a0)
-534 DisposeRegion(region)(a0)
-540 FreeVPortCopLists(vp)(a0)
-546 FreeCopList(copList)(a0)
-552 ClipBlit(srcRP,xSrc,ySrc,destRP,xDest,yDest,xSize,ySize,minterm)(a0,d0/d1/a1,d2/d3/d4/d5/d6)
-558 XorRectRegion(region,rectangle)(a0/a1)
-564 FreeCprList(cprList)(a0)
-570 GetColorMap(entries)(d0)
-576 FreeColorMap(colorMap)(a0)
-582 GetRGB4(colorMap,entry)(a0,d0)
-588 ScrollVPort(vp)(a0)
-594 UCopperListInit(uCopList,n)(a0,d0)
-600 FreeGBuffers(anOb,rp,flag)(a0/a1,d0)
-606 BltBitMapRastPort(srcBM,x,y,destRP,x,y,WId,Height,minterm)(a0,d0/d1/a1,d2/d3/d4/d5/d6)
-612 OrRegionRegion(srcRegion,destRegion)(a0/a1)
-618 XorRegionRegion(srcRegion,destRegion)(a0/a1)
-624 AndRegionRegion(srcRegion,destRegion)(a0/a1)
-630 SetRGB4CM(colorMap,index,red,green,blue)(a0,d0/d1/d2/d3)
-636 BltMaskBitMapRastPort
(srcBM,x,y,destRP,x,y,Wid,High,mterm,Mask)(a0,d0/d1/a1,d2/d3/d4/d5/d6/a2)
-642 RESERVED
-648 RESERVED
-654 AttemptLockLayerRom(layer)(a5)

*** functions in Release 2.0 or higher ***

-660 GfxNew(gfxNodeType)(d0)
-666 GfxFree(gfxNodePtr)(a0)
-672 GfxAssociate(associateNode,gfxNodePtr)(a0/a1)
-678 BitMapScale(bitScaleArgs)(a0)
-684 ScalerDiv(factor,numerator,denominator)(d0/d1/d2)
-690 TextFit
(rp,string,strLen,textExtent,constrainingExtent,strDirection,constrainingBitWidth,constrainingBitHeight)(
a1,a0,d0/a2)

INTUITION

-30 OpenIntuition()()
-36 Intuition(iEvent)(a0)
-42 AddGadget(window,gadget,position)(a0/a1,d0)
-48 ClearDMRequest(window)(a0)
-54 ClearMenuStrip(window)(a0)
-60 ClearPointer(window)(a0)
-66 CloseScreen(screen)(a0)

-72 CloseWindow(window)(a0)
-78 CloseWorkBench()()
-84 CurrentTime(seconds,micros)(a0/a1)
-90 DisplayAlert(alertNumber,string,height)(d0/a0,d1)
-96 DisplayBeep(screen)(a0)
-102 DoubleClick(sSeconds,sMicros,cSeconds,cMicros)(d0/d1/d2/d3)
-108 DrawBorder(rp,border,leftOffset,topOffset)(a0/a1,d0/d1)
-114 DrawImage(rp,image,leftOffset,topOffset)(a0/a1,d0/d1)
-120 EndRequest(requester,window)(a0/a1)
-126 GetDefPrefs(preferences,size)(a0,d0)
-132 GetPrefs(preferences,size)(a0,d0)
-138 InitRequester(requester)(a0)
-144 ItemAddress(menuStrip,menuNumber)(a0,d0)
-150 ModifyIDCMP(window,flags)(a0,d0)
-156 ModifyProp
(gadget,window,requester,flags,horizPot,vertPot,horizBody,vertBody)(a0/a1/a2,d0/d1/d2/d3/d4)
-162 MoveScreen(screen,dx,dy)(a0,d0/d1)
-168 MoveWindow(window,dx,dy)(a0,d0/d1)
-174 OffGadget(gadget,window,requester)(a0/a1/a2)
-180 OffMenu(window,menuNumber)(a0,d0)
-186 OnGadget(gadget,window,requester)(a0/a1/a2)
-192 OnMenu(window,menuNumber)(a0,d0)
-198 OpenScreen(newScreen)(a0)
-204 OpenWindow(newWindow)(a0)
-210 OpenWorkBench()()
-216 PrintIText(rp,iText,left,top)(a0/a1,d0/d1)
-222 RefreshGadgets(gadgets,window,requester)(a0/a1/a2)
-228 RemoveGadget(window,gadget)(a0/a1)
-234 ReportMouse(flag,window)(d0/a0)
-240 Request(requester,window)(a0/a1)
-246 ScreenToBack(screen)(a0)
-252 ScreenToFront(screen)(a0)
-258 SetDMRequest(window,requester)(a0/a1)
-264 SetMenuStrip(window,menu)(a0/a1)
-270 SetPointer(window,pointer,height,width,xOffset,yOffset)(a0/a1,d0/d1/d2/d3)
-276 SetWindowTitles(window,windowTitle,screenTitle)(a0/a1/a2)
-282 ShowTitle(screen,showIt)(a0,d0)
-288 SizeWindow(window,dx,dy)(a0,d0/d1)
-294 ViewAddress()()
-300 ViewPortAddress(window)(a0)
-306 WindowToBack(window)(a0)
-312 WindowToFront(window)(a0)
-318 WindowLimits(window,widthMin,heightMin,widthMax,heightMax)(a0,d0/d1/d2/d3)
-324 SetPrefs(preferences,size,inform)(a0,d0/d1)

-330 IntuiTextLength(iText)(a0)
-336 WBenchToBack()()
-342 WBenchToFront()()
-348 AutoRequest(window,body,posText,negText,pFlag,nFlag,width,height)(a0/a1/a2/a3,d0/d1/d2/d3)
-354 BeginRefresh(window)(a0)
-360 BuildSysRequest(window,body,posText,negText,flags,width,height)(a0/a1/a2/a3,d0/d1/d2)
-366 EndRefresh(window,complete)(a0,d0)
-372 FreeSysRequest(window)(a0)
-378 MakeScreen(screen)(a0)
-384 RemakeDisplay()()
-390 RethinkDisplay()()
-396 AllocRemember(rememberKey,size,flags)(a0,d0/d1)
-402 AlohaWorkbench(wbport)(a0)
-408 FreeRemember(rememberKey,reallyForget)(a0,d0)

-414 LockIBase(dontknow)(d0)
-420 UnlockIBase(ibLock)(a0)

*** functions in Release 1.2 or higher ***

-426 GetScreenData(buffer,size,type,screen)(a0,d0/d1/a1)
-432 RefreshGList(gadgets,window,requester,numGad)(a0/a1/a2,d0)
-438 AddGList(window,gadget,position,numGad,requester)(a0/a1,d0/d1/a2)
-444 RemoveGList(remPtr,gadget,numGad)(a0/a1,d0)
-450 ActivateWindow(window)(a0)
-456 RefreshWindowFrame(window)(a0)
-462 ActivateGadget(gadgets,window,requester)(a0/a1/a2)
-468 NewModifyProp
(gadget,window,requester,flags,horizPot,vertPot,horizBody,vertBody,numGad)
(a0/a1/a2,d0/d1/d2/d3/d4/d5)

*** functions in Release 2.0 or higher ***

-474 QueryOverscan(displayID,rect,oScanType)(a0/a1,d0)
-480 MoveWindowInFrontOf(window,behindWindow)(a0/a1)
-486 ChangeWindowBox(window,left,top,width,height)(a0,d0/d1/d2/d3)
-492 SetEditHook(hook)(a0)
-498 SetMouseQueue(window,queueLength)(a0,d0)
-504 ZipWindow(window)(a0)

--- public screens ---

-510 LockPubScreen(name)(a0)
-516 UnlockPubScreen(name,screen)(a0/a1)
-522 LockPubScreenList()()
-528 UnlockPubScreenList()()
-534 NextPubScreen(screen,namebuf)(a0/a1)
-540 SetDefaultPubScreen(name)(a0)
-546 SetPubScreenModes(modes)(d0)
-552 PubScreenStatus(screen,statusFlags)(a0,d0)
-558 ObtainGIRPort(gInfo)(a0)
-564 ReleaseGIRPort(rp)(a0)
-570 GadgetMouse(gadget,gInfo,mousePoint)(a0/a1/a2)
-576 *intuitionPrivate1()()
-582 GetDefaultPubScreen(nameBuffer)(a0)
-588 EasyRequestArgs(window,easyStruct,idcmpPtr,args)(a0/a1/a2/a3)
-594 BuildEasyRequestArgs(window,easyStruct,idcmp,args)(a0/a1,d0/a3)
-600 SysReqHandler(window,idcmpPtr,waitInput)(a0/a1,d0)
-606 OpenWindowTagList(newWindow,tagList)(a0/a1)
-612 OpenScreenTagList(newScreen,tagList)(a0/a1)

---new Image functions---

-618 DrawImageState(rp,image,leftOffset,topOffset,state,drawInfo)(a0/a1,d0/d1/d2/a2)
-624 PointInImage(point,image)(d0/a0)
-630 EraseImage(rp,image,leftOffset,topOffset)(a0/a1,d0/d1)
-636 NewObjectA(classPtr,classID,tagList)(a0/a1/a2)
-642 DisposeObject(object)(a0)
-648 SetAttrsA(object,tagList)(a0/a1)
-654 GetAttr(attrID,object,storagePtr)(d0/a0/a1)

---special set attribute call for gadgets---

-660 SetGadgetAttrsA(gadget,window,requester,tagList)(a0/a1/a2/a3)

-666 NextObject(objectPtrPtr)(a0)
-672 *intuitionPrivate2()()
-678 MakeClass(classID,superClassID,superClassPtr,instanceSize,flags)(a0/a1/a2,d0/d1)
-684 AddClass(classPtr)(a0)
-690 GetScreenDrawInfo(screen)(a0)
-696 FreeScreenDrawInfo(screen,drawInfo)(a0/a1)
-702 ResetMenuStrip(window,menu)(a0/a1)
-708 RemoveClass(classPtr)(a0)
-714 FreeClass(classPtr)(a0)
-720 *intuitionPrivate3()()
-726 *intuitionPrivate4()()

DISKFONT

-30 OpenDiskFont(textAttr)(a0)
-36 AvailFonts(buffer,bufBytes,flags)(a0,d0/d1)

*** functions in Release 1.2 or higher ***

-42 NewFontContents(fontsLock,fontName)(a0/a1)
-48 DisposeFontContents(fontContentsHeader)(a1)

*** functions in Release 2.0 or higher ***

-54 NewScaledDiskFont(sourceFont,destTextAttr)(a0/a1)

Amiga Hardware Registers

The following are a list of memory locations where direct access to the Agnus, Denise and Paula chips is possible. It is illegal to access any of these registers if you wish your program to behave correctly in the Amiga environment. However in BlitzMode most of these registers may be accessed taking into consideration the accompanying documentation.

An * next to any description states that the option is available only with the new ECS (Enhanced Chip Set).

Also note that any reference to memory pointers MUST point to chip mem as the Amiga Chip Set is NOT capable of accessing FAST mem. This includes BitPlane data, copper lists, Sprite Data, Sound DATA etc. etc.

BitPlane & Display Control

The Amiga has great flexibility in displaying graphics at different resolutions and positions on the monitor. The hardware registers associated with the display are nearly always loaded by the copper and not with the 68000 processor.

```
#BPLCON0=$100
#BPLCON1=$102
#BPLCON2=$104
#BPLCON3=$106 ; (ECS only)
```

BIT#	BPLCON0	BPLCON1	BPLCON2
15	HIRES (70ns pixles)		
14	BPU2 \		
13	BPU1 I #BitPlanes(0-6)		
12	BPU0 /		
11	HOMOD Hold & Modify		
10	DBLPF DualPlayField		
09	COLOR Composite Enable		
08	GAUD GenlockAudio		
07			PF2H3\
06	*SHRES SuperHires	PF2H2 I Playfield 2	PF2PRI DBLPF Priority
05	*BPLHWRM	PF2H1 I horizontal	PF2P2
04	*SPRHWRM	PF2H0/ scroll	PF2P1 Priority to sprites
03	LPEN LightPenEnable	PF1H3\	PF2P0
02	LACE Interlace	PF1H2 I Playfield 1	PF1P2
01	ERSY ExternalSync	PF1H1 I Horizontal	PF1P1 Priority to sprites
00		PF1H0/ scroll	PF1P0

```
#BPL0PTH=$E0    ;BitPlane Pointer 0 High Word
#BPL0PTL=$E2    ;BitPlane Pointer 0 Low Word
#BPL1PTH=$E4
#BPL1PTL=$E6
#BPL2PTH=$E8
#BPL2PTL=$EA
#BPL3PTH=$EC
#BPL3PTL=$EE
#BPL4PTH=$F0
#BPL4PTL=$F2
#BPL5PTH=$F4
#BPL5PTL=$F6
```

Each pair of registers contain an 18 bit pointer to the address of BitPlanex data in chip memory. They MUST be reset every frame usually by the copper.

```
#BPL1MOD=$108  ;Bitplane Modulo for Odd Planes
#BPL2MOD=$10A  ;Bitplane Modulo for EvenPlanes
```

At the end of each display line, the BPLxMODs are added to the the BitPLane Pointers so they point to the address of the next line.

```
#DIWSTOP=$090  ; display window stop
#DIWSTRT=$08E  ; display window start
```

These two registers control the display window size and position. The following bits are assigned

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

For DIWSTRT V8=0 & H8=0 restricting it to the upper left of the screen. For DIWSTOP V8=1 & H8=1 restricting it to the lower right of the screen.

```
#DDFSTOP= $094  ; data fetch stop
#DDFSTRT=$092   ; data fetch start
```

The two display data fetch registers control when and how many words are fetched from the bitplane for each line of display.

Typical values are as follows:

lores 320 pixels, DDFSTRT & DDFSTOP = $38 & $D0
hires 640 pixels, DDFSTRT & DDFSTOP = $3C & $d4

If smooth scrolling is enabled DDFSTRT should be 2 less than above.

```
#BPL1DAT $110  ; BitPlane Data parallel to serial converters
#BPL2DAT $112
#BPL3DAT $114
#BPL4DAT $116
#BPL5DAT $118
#BPL6DAT $11A
```

These 6 registers receive the DMA data fetched by the BitPlane engine, and output it serially to the Amiga DACS, triggered by writing to BPL1DAT. Not intended for programmer access.

The Copper

The Copper is found on the Agnus chip, it's main job is to 'poke' values into the hardware registers in sync with the video beam. The main registers it updates are BitPlane ptrs, Sprites and other control words that HAVE to be reset every frame. It's also used to split the screen vertically as it is capable of waiting for certain video beam positions before writing data. Its also capable of waiting for the blitter to finish as well as skipping instructions if beam position is equal to certain values.

#COP1LCH=$080
#COP1LCL=$082

#COP2LCH=$084
#COP2LCL=$086

Each pair of registers contain an 18 bit pointer to the address of a Copper List in chip mem. The Copper will automatically jump to the address in COP1 at the beginning of the frame and is able to jump to COP2 if the following strobe is written to.

#COPJMP1=$88
#COPJMP2=$8A

When written to these addresses cause the copper to jump to the locations held in COP1LC & COP2LC. The Copper can write to these registers itself causing its own indirect jump.

#COPCON=$2E

By setting bit 1 of this register the copper is allowed to access the blitter hardware.

The copper fetches two words for each instruction from its current copper list. The three instructions it can perform and their relevant bits are as follows:

Bit#	MOVE		WAIT UNTIL		SKIP IF	
15	x	RD15	VP7	BFD	VP7	BFD
14	x	RD14	VP6	VE6	VP6	VE6
13	x	RD13	VP5	VE5	VP5	VE5
12	x	RD12	VP4	VE4	VP4	VE4
11	x	RD11	VP3	VE3	VP3	VE3
10	x	RD10	VP2	VE2	VP2	VE2
09	x	RD09	VP1	VE1	VP1	VE1
08	DA8	RD08	VP0	VE0	VP0	VE0
07	DA7	RD07	HP8	HE8	HP8	HE8
06	DA6	RD06	HP7	HE7	HP7	HE7
05	DA5	RD05	HP6	HE6	HP6	HE6
04	DA4	RD04	HP5	HE5	HP5	HE5
03	DA3	RD03	HP4	HE4	HP4	HE4
02	DA2	RD02	HP3	HE3	HP3	HE3
01	DA1	RD01	HP2	HE2	HP2	HE2
00	0	RD00	1	0	1	1

The MOVE instruction shifts the value held in RD15-0 to the destination address calculated by $DFF000 +DA8-1.

The WAIT UNTIL instruction places the copper in a wait state until the video beam position is past HP,VP (xy coordinates). The Copper first logical ANDS (masks) the video beam with HE,VE before doing the comparison. If BFD is set then the blitter must also be finished before the copper will exit its wait state.

The SKIP IF instruction is similar to the WAIT UNTIL instruction but instead of placing the copper in a wait state if the video beam position fails the comparison test it skips the next MOVE instruction.

A detailed discussion of creating copper lists in included in the Blitz 2 user guide.

Colour Registers

The following 32 color registers can each represent one of 4096 colors.

#COLOR00=$180	#COLOR08=$190	#COLOR16=$1A0	#COLOR24=$1B0
#COLOR01=$182	#COLOR09=$192	#COLOR17=$1A2	#COLOR25=$1B2
#COLOR02=$184	#COLOR10=$194	#COLOR18=$1A4	#COLOR26=$1B4
#COLOR03=$186	#COLOR11=$196	#COLOR19=$1A6	#COLOR27=$1B6
#COLOR04=$188	#COLOR12=$198	#COLOR20=$1A8	#COLOR28=$1B8
#COLOR05=$18A	#COLOR13=$19A	#COLOR21=$1AA	#COLOR29=$1BA
#COLOR06=$18C	#COLOR14=$19C	#COLOR22=$1AC	# COLOR30=$1BC
#COLOR07=$18E	#COLOR15=$19E	#COLOR23=$1AE	#COLOR31=$1BE

The bit usage for each of the 32 colors is:

BIT#	15 14 13 12	11 10 09 08	07 06 05 04	03 02 01 00
	x x x x	R3 R2 R1 R0	G3 G2 G1 G0	B3 B2 B1 B0

This represents a combination of 16 shades of red, green and blue.

Blitter Control

The Blitter is located on the Agnus, it's main function is to move blocks of data around chip mem. It has 3 input channels A,B & C and 1 output channel D. A simple block move would use 1 input channel and the 1 output channel, taking 4 clock ticks per cycle. A complex move such as a moving a shape to a destination with a cookie cut would use all 3 input channels and the output channel taking 8 clock ticks per cycle.

The main parameters of the blitter include the width and height of the
block to be moved (width is in multiples of words), a start address for each channel, a modulo for each channel that is added to there address at the end of each line so they point to the next line, a logic function that specifies which input channels data will be sent to the destination channel.

Logic Function Calculation.

The following is a table to work out the logic function (known as the minterm) for a blitter operation.

A	B	C	D
0	0	0	LF0
0	0	1	LF1
0	1	0	LF2
0	1	1	LF3
1	0	0	LF4
1	0	1	LF5
1	1	0	LF6
1	1	1	LF7

If the Blitter is set up so that channel A points to the cookie, B points to the shape to be copied and C&D point to the destination bitplane (such as how Blitz 2 uses the blitter) we would specify the following conditions:

When A is 1 then make D=B
When A is 0 then make D=C

Using the above table we calculate the values of LF0-LF7 when these two conditions are met. The top line has A=0 so LF0 becomes the value in the C column which is a 0. A is 0 in the first 4 rows so LF0-LF3 all reflect the bits in the C column (0101) and A=1 in the lower 4 rows so LF4-LF7 reflect the bits in the B column (0011).

This generates a minterm LF0-LF7 of %10101100 or in hex $AC.

Note: read the values of LF7 to LF0 from bottom to top to calculate the correct hexadecimal minterm.

```
#BLTAPTH=$50
#BLTAPTL=$52

#BLTBPTH=$4C
#BLTBPTL=$4E

#BLTCPTH=$48
#BLTCPTL=$4A

#BLTDPTH=$54
#BLTDPTL=$56
```

Each pair of registers contain an 18 bit pointer to the start address of the 4 blitter channels in chip mem.

```
#BLTAMOD=$64
#BLTBMOD=$62
#BLTCMOD=$60
#BLTDMOD=$66
```

The 4 modulo values are added to the blitter pointers at the end of each line.

```
#BLTADAT=$74
#BLTBDAT=$72
#BLTCDAT= $70
```

If a blitter channel is disabled the BLTxDAT register can be loaded with a constant value which will remain unchanged during the blit operation.

```
#BLTAFWM=$44  ; Blitter first word mask for source A
#BLTALWM=$46  ; Blitter last word mask for source A
```

During a Blitter operation these two registers are used to mask the contents of BLTADAT for the first and last word of every line.

```
#BLTCON0=$100
#BLTCON1=$102
```

The following bits in BLTCON0 & BLTCON1 are as follows.

BIT#	BLTCON0	BLTCON1
15	ASH3	BSH3
14	ASH2	BSH2
13	ASH1	BSH1
12	ASH0	BSH0
11	USEA	x
10	USEB	x
09	USEC	x
08	USED	x
07	LF7	x
06	LF6	x
05	LF5	x
04	LF4	EFE
03	LF3	IFE
02	LF2	FCI
01	LF1	DESC
00	LF0	0 (1=line mode)

ASH is the amount that source A is shifted (barrel rolled)
USEx enables each of the 4 blitter channels
LF holds the logic function as discussed previously in this section
BSH is the amount that source B is shifted (barrel rolled)
EFE is the Exclusive Fill Enable flag
IFE is the Inclusive Fill Enable flag
FCI is the Fill Carry Input
DESC is the descending flag (blitter uses decreasing addressing)

```
#BLTSIZE=$58
```

By writing the height and width of the blit operation to BLTSIZE the the blitter will start the operation. Maximum size is 1024 high and 64 words (1024 bits) wide. The following defines bits in BLITZSIZE

BIT#	15 14 13 12 11 10 09 08 07 06	05 04 03 02 01 00
	h9 h8 h7 h6 h5 h4 h3 h2 h1 h0	w5 w4 w3 w2 w1 w0

```
#BLTSIZV= $5C  ;(ECS ONLY)
#BLTSIZH =$5C  ;(ECS ONLY)
```

With the new ECS writing to BLTSIZV first and then BLTSZH the blitter can operate on blocks as large as 32K x 32K pixels in size.

The Blitter is also able to perform linedrawing and filled polygon functions. Details about using the blitter for these functions can be found on the examples disk included with Blitz 2.

Audio Control

The Amiga has 4 channels of 8 bit audio, each with their own memory access, period and volume control. The following are a list of the applicable hardware registers.

```
#AUD0LCH=$A0  ;pairs of 24 bit memory pointers to audio data in chip mem
#AUD0LCL=$A2
#AUD1LCH=$B0
#AUD1LCL=$B2
#AUD2LCH=$C0
#AUD2LCL=$C2
#AUD3LCH=$D0
#AUD3LCL=$D2

#AUD0LEN=$A4  ;volume registers (0-63)
#AUD1LEN=$B4
#AUD2LEN=$C4
#AUD3LEN=$D4

#AUD0PER=$A6  ;period
#AUD1PER=$B6
#AUD2PER=$C6
#AUD3PER=$D6

#AUD0VOL=$A8
#AUD1VOL=$B8
#AUD2VOL=$C8
#AUD3VOL=$D8

#AUD0DAT=$AA
#AUD1DAT=$BA
#AUD2DAT=$CA
#AUD3DAT=$DA
```

Sprite Control

The Amiga hardware is capable of displaying eight 4 colour sprites or four 16 colour sprites. Standard control of sprites is done by using the copper to setup the 8 sprite pointers at the beginning of each frame.

```
#SPR0PTH=$120  ;pairs of 24 bit memory pointers to sprite data in chip mem
#SPR0PTL=$122
#SPR1PTH=$124
#SPR1PTL=$126
#SPR2PTH=$128
#SPR2PTL=$12A
#SPR3PTH=$12C
```

#SPR3PTL=$12E
#SPR4PTH=$130
#SPR4PTL=$132
#SPR5PTH=$134
#SPR5PTL=$136
#SPR6PTH=$138
#SPR6PTL=$13A
#SPR7PTH=$13C
#SPR7PTL=$13E

The pointers should point to data that is begins with two words containing the SPRPOS & SPRCTL values for that sprite, followed by its image data and with two null words that terminate the data.

#SPR0POS = $140 #SPR0CTL = $142 #SPR0DATA = $144 #SPR0DATB = $146
#SPR1POS = $148 #SPR1CTL = $14A #SPR1DATA = $14C #SPR1DATB = $14E
#SPR2POS = $150 #SPR2CTL = $152 #SPR2DATA = $154 #SPR2DATB = $156
#SPR3POS = $158 #SPR3CTL = $15A #SPR3DATA = $15C #SPR3DATB = $15E
#SPR4POS = $160 #SPR4CTL = $162 #SPR4DATA = $164 #SPR4DATB = $166
#SPR5POS = $168 #SPR5CTL = $16A #SPR5DATA = $16C #SPR5DATB = $16E
#SPR6POS = $170 #SPR6CTL = $172 #SPR6DATA = $174 #SPR6DATB = $176
#SPR7POS = $178 #SPR7CTL = $17A #SPR7DATA = $17C #SPR7DATB = $17E

Using standard sprite DMA the above registers are all loaded from the sprite data pointed to in chip mem by the sprite pointers. These registers are only of interest to people wanting to 'multiplex' sprites by using the copper to load these registers rather than sprite DMA.

The following is bit definitions of both SPRPOS and SPRCTL.

BIT#	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
SPRPOS	SV7	SV6	SV5	SV4	SV3	SV2	SV1	SV0	SH8	SH7	SH6	SH5	SH4	SH3	SH2	SH1
SPRCTL	EV7	EV6	EV5	EV4	EV3	EV2	EV1	EV0	ATT	X	X	X	X	SV8	EV8	SH0

SV is the vertical start position of the sprite
SH is the horizontal position of the sprite (calulated in lores pixels only)
EV is the end vertical position
ATT is the sprite attached bit (connects odd sprites to their predecessors)

Interupt Control

#INTENA=$9A ;interupt enable write address
#INTENAR=$1C ;interupt enable read address

#INTREQ=$9C ;interupt request write address
#INTREQR=$9C ;interupt request read address

INTENA is used to enable or disable interupts. If the value written to INTENA has bit 15 set any other of the bits enable their corresponding interupts. If bit 15 is clear any of the other bits set will disable their corresponding interupts.

INTENAR will return which interupts are currently enabled.

INTREQ is used to initiate or clear an interupt. It is mostly used to clear the interupt by the interupt handler. Again Bit# 15 states whether the corrsponding interupts will be requested or cleared.

INTREQR returns which interupts are currently requested.

The following bit definitions relate to the 4 interupt control registers.

BIT#	NAME	LEVEL	DESCRIPTION
15	SET/CLR		determines if bits written with 1 are set or cleared
14	INTEN		master interupt enable
13	EXTER	6	external interupt
12	DSKSYN	5	disk sync register (same as DSKSYNC)
11	RBF	5	serial port Receive Buffer Full
10	AUD3	4	audio channel 3 finished
09	AUD2	4	audio channel 2 finished
08	AUD1	4	audio channel 1 finished
07	AUD0	4	audio channel 0 finished
06	BLIT	3	blitter finished
05	VERTB	3	start of vertical blank interupt
04	COPER	3	copper
03	PORTS	2	I/O ports and timers
02	SOFT	1	reserved for software initiated interupts
01	DSKBLK	1	disk block finished
00	TBE	1	serial port Transmit Buffer Empty

The following locations hold the address of the 68000 interupt handler code in memory for each level of interupt.

LEVEL	68000 Address
6	$78
5	$74
4	$70
3	$6c
2	$68
1	$64

DMA Control

DMA stands for direct memory access. Chip mem can be accessed by the display, blitter, copper, audio, sprites and diskdrive without using the 68000 processor. DMACON enables the user to lock out any of these from having direct memory access (DMA) to chipmem.

As with INTENA bit 15 of DMACON signals whether the write opertaion should clear or set the relevant bits of the DMA control.

DMACONR will not only return which channels have DMA access but has flags BBUSY which return true if the blitter is in operation and BZERO which return if the Blitter has generated any 1's from it logic function (useful for collision detection etc.)

```
#DMACON=$96     ;DMA control write (clear or set)
#DMACONR=$02    ;DMA control read (and blitter status) read
```

The following are the bits assigned to the two DMACON registers.

BIT#	NAME	DESCRIPTION
15	SET/CLR	determines if bits written with 1 are set or cleared
14	BBUSY	blitter busy flag
13	BZERO	blitter logic zero
12	X	
11	X	
10	BLTPRI	"blitter nasty" signals blitter has DMA priority over CPU
09	DMAEN	enable all DMA below
08	BPLEN	BitPlane DMA enable
07	COPEN	Copper DMA enable
06	BLTEN	Blitter DMA enable
05	SPREN	Sprite DMA enable
04	DSKEN	Disk DMA enable
03	AUD3EN	Audio channel 3 DMA enable
02	AUD2EN	Audio channel 2 DMA enable
01	AUD1EN	Audio channel 1 DMA enable
00	AUD0EN	Audio channel 0 DMA enable

Miscelaneous Amiga Chip Locations

The following is a list of the other $dff000 addresses not covered by the previous sections. Because of their complex nature other texts should be referred to for more information.

```
#ADKCON=$09E        ;Audio/Disk control write
#ADKCONR=$010       ;Audio/Disk control read
#BEAMCON0=$1DC      ;ECS Beam Counter Control Register
#CLXCON=$098        ;Collision control register (see Blitz)( collision commands)
#CLXDAT=$00E        ;Collision data register (see Blitz)( collision commands)

#DENISEID=$07c      ;ECS Denise chip revsion level
#DIWHIGH=$1E4       ;ECS display window high
#DSKBYTER=$01A      ;disk data byte and status read
#DISKDAT=$026       ;disk DMA data write
#DISKDATR=$008      ;disk DMA data read
#DSKLEN=$024        ;disk length
#DSKPTH=$020        ;disk pointer high
#DSKPTL=$022        ;disk pointer low
#DSKSYNC=$07e       ;disk sync register

#HBSTOP=$1C6        ;ECS horizontal line position for HBLANK stop
#HBSTRT=$1C4        ;ECS horizontal line position for HBLANK start
#HCENTER=$1E2       ;ECS horizontal position for Vsync on interlace
#HSSTOP=$1C2        ;ECS horizontal line position for HSYNC stop
#HSSTRT=$1DE        ;ECS horizontal line position for HSYNC strt
#HTOTAL=$1C0        ;ECS highest number count for horizontal line
```

#JOY0DAT=$00A	;joystick mouse data left up/dwn
#JOY1DAT=$00C	;joystick mouse data right up/dwn
#JOYTEST=$036	;mouse counters write
#POT0DAT=$012	;pot counter data left pair
#POT1DAT=$014	;pot counter data right pair
#POTGO=$034	;pot port data write and start
#POTGOR=$016	;pot port data read
#REFPTR=$028	;refresh pointer
#SERDAT=$030	;serial port data write (with stop bit)
#SERDATR=$018	;serial port data read and status bits
#SERPER=$032	;baud rate and 9 bit word flag
#STREQU=$038	;strobe for horizontal sync with VB and EQU
#STRHOR=$03C	;strobe for horizontal sync
#STRLONG=$03E	;ECS strobe for id of long horizontal line
#STRVBL=$03A	;strobe for horizontal sync with VB
#VBSTOP=$1CE	;ECS vertical line for vblank stop
#VBSTRT=$1CC	;ECS vertical line for vblank start
#VHPOSR=$006	;video beam position
#VHPOSW=$02C	;write vertical beam position
#VPOSR=$004	;video beam position (vertical most significant bit)
#VPOSW=$02A	;write vertical beam position MSB
#VSSTOP=$1CA	;ECS vertical line position for VSYNC stop
#VSSTRT=$1E0	;ECS vertical line position for VSYNC start
#VTOTAL=$1C8	;ECS highest numbered vertical line

Amiga CIAs

The Amiga has two 8520 Complex Interface Adapter (CIA) which handle most of the Amiga I/O activities. Note that each register should be accessed as a byte and NOT a word. The following is an address map of both Amiga CIAs.

Byte Address	Register	b7	b6	b5	b4	b3	b2	b1	b0
$BFE001	pra	FIR1	FIR0	RDY	TK0	WPR0	CHNG	LED	OVL
$BFE101	prb	Parallel Port							
$BFE201	ddra	Direction for Port A (1=output)							
$BFE301	ddrb	Direction for Port B (1=output)							
$BFE401	talo	Timer A High Byte							
$BFE501	tahi	Timer A High Byte							
$BFE601	tblo	Timer B Low Byte							
$BFE701	tbhi	Timer B High Byte							
$BFE801	todlo	50/60 Hz Event Counter bits 7-0							
$BFE901	todmid	50/60 Hz Event Counter bits 15-8							
$BFEA01	todhi	50/60 Hz Event Counter bits 23-16							
$BFEB01		not used							
$BFEC01	sdr	Serial Data Register (connected to keyboard)							
$BFED01	icr	Interrupt Control Register							
$BFEE01	cra	Control Register A							
$BFEF01	crb	Control Register B							

CIA B

Address	Register	b7 b6 b5 b4 b3 b2 b1 b0
$BFD000	pra	DTR RTS CD CTS DSR SEL POUT BUSY
$BFD100	prb	MTR SEL3 SEL2 SEL1 SEL0 SIDE DIR STEP
$BFD200	ddra	Direction for Port A (1=output)
$BFD300	ddrb	Direction for Port B (1=output)
$BFD400	talo	Timer A High Byte
$BFD500	tahi	Timer A High Byte
$BFD600	tblo	Timer B Low Byte
$BFD700	tbhi	Timer B High Byte
$BFD800	todlo	Horizontal Sync Event Counter bits 7-0
$BFD900	todmid	Horizontal Sync Event Counter bits 15-8
$BFDA00	todhi	Horizontal Sync Event Counter bits 23-16
$BFDB00		not used
$BFDC00	sdr	Serial Data Register (connected to keyboard)
$BFDD00	icr	Interrupt Control Register
$BFDE00	cra	Control Register A
$BFDF00	crb	Control Register B

68000 Assembly Language

Although Blitz 2 is a BASIC compiler, it also has an 'inline assembler' and can be used as a fully fledged assembler. Assembly language is the language of the microprocessor, in the case of the Amiga, the 68000 microprocessor.

The following is a brief description of the Motorola 68000 microprocessor and its instruction set, for more information we recommend the data books published by Motorola themselves as the best source of reference material.

Registers

The 68000 has 16 internal registers, these may be thought of as high speed variables each capable of storing a long word (32 bits). The 8 data registers are used mainly for calculations while the 8 address registers are mostly used for pointing to locations in memory.

The registers are named D0-D7 and A0-A7. The 68000 also has several specialised registers, the program counter (PC) and the status register (SR). The program counter points to the current instruction that the microprocessor is executing, while the status register is a bunch of flags with various meanings.

Addressing

The main job of the microprocessor is to read information from memory, perform a calculation and then write the result back to memory.

For the processor to access memory it has to generate a memory address for the location it wishes to access (read or write to). The following are the different ways the 68000 can generate addresses.

Register Direct

```
MOVE d1,d0
```

The actual value in the register d1 is copied into d0

Address Register Indirect

```
MOVE (a0),d0
```

a0 is a pointer to somewhere in memory. The value at at this location is copied into the register d0.

Address Register Indirect with Postincrement

```
MOVE (a0)+,d0
```

The value at the location pointed to by a0 is copied into the register d0, then a0 is incremented so it points to the next memory location.

Address Register Indirect with Predecrement

```
MOVE -(a0),d0
```

a0 is first decremented to point to the memory location before the one it currently points to then the value at the new memory location is copied into d0.

Address Register Indirect with Displacement

```
MOVE 16(a0),d0
```

The memory location located 16 bytes after that which is pointed to by address register a0 is copied to d0.

Address Register Indirect with Index

```
MOVE 16(a0,d1),d0
```

The memory location is calculated by adding the contents of a0 with d1 plus 16.

Absolute Address

```
MOVE $dff096,d0
```

The memory location $dff096 is used.

Program Counter with Displacement

```
MOVE label(pc),d0
```

This is the same as absolute addressing but because the memory address is an offset from the program counter (no bigger than 32000 bytes) it is MUCH quicker.

Program Counter with Index

```
MOVE label(pc,d1),d0
```

The address is calulated as the location of label plus the contents of data register d1.

Immediate Data

```
MOVE #20,d0
```

The value 20 is moved to the data register.

Program Flow

As mentioned previously the microprocessor has a special register known as the program counter that points to the next instruction to be executed. By changing the value in the program counter a 'goto' can be performed. The JMP instruction load the program counter with a new value, it supports most of the addressing modes.

A branch is a program counter relative form of the JMP instruction. Branches can also be performed on certain conditions such as BCC which will only cause the program flow to change if the Carry flag in the status register is currently set.

A 'gosub' can be prformed using the JSR and BSR commands. The current value of the program counter is remembered on the stack before the jump or branch is performed. The RTS command is used to 'return' to the original program location.

The Stack

The Amiga sets aside a certain amount of memory for each task known as a stack. The address register A7 is used to point to the stack and should never be used as a general purpose address register.

The 68000 uses predecrement addressing to push data onto the stack and postincrement addressing to pull information off the stack.

JSR is the same as MOVE.l pc,-(a7) and then JMP

RTS is the same as MOVE.l (a7)+,pc

The stack can be used to temporarily store internal registers. To save and restore all the 68000 registers the following code is often used

```
ASubroutine:
   MOVEM.l d0-d7/a0-a6,-(a7)   ;push all register on stack
   ;main subroutine code here which can stuff up registers without worrying
   MOVEM.l (a7)+,d0-d7/a0-a6   ;pull registers off stack
   RTS                         ;returnfrom subroutine
```

Condition Flags

The status register is a special 68000 register that holds, besides other things all the condition codes. The following are a list of the condition flags:

Code	Name	Meaning
N	negative	reflects the most significant bit of the result of the last operation.
Z	zero	is set if the result is zero, cleared otherwise.
C	carry	is set when an add, subtract or compare operation generate a carry
X	extend	is a mirror of the carry flag, however its not affected by data movement.
V	overflow	is set when an aritmeitic operation causes an overflow, a situation where the operand is not large enough to represent the result.

Conditional Tests

Branches and Sets can be performed conditionally. The following is a list of the possible conditions that can be tested before a branch or set is performed.

cc	condition	coding	test
T	true	0000	1
F	false	0001	0
HI	high	0010	not C & not Z
LS	lowsam	0011	C I Z
CC	carry clr	0100	not C
CS	carry set	0101	C
NE	ot equal	0110	not Z
EQ	equal	0111	Z
VC	overflow clr	1000	not V
VS	overflow set	1001	V
PL	plus	1010	not N
MI	minus	1011	N
GE	greater equal	1100	N&V I notN&notV
LT	less than	1101	N&notV I notN&V
GT	greater than	1110	N&V&notZ I notN&notV&notC
LE	less or equal	1111	Z I N&notV I notN&V

Operand Sizes

The 68000 can perform operations on bytes, words and long words. By adding a suffix .b .w or .l to the opcode, the assembler knows which data size you wish to use, if no suffix is present the word size is default. There is no speed increase using bytes instead of words as the 68000 is a 16 bit microprocessor and so no overhead is needed for 16 bit operations. However 32 bit long words do cause overhead with extra read and write cycles needed to perform operations on a bus that can only handle 16 bits at a time.

The 68000 Instruction Set

The following is a brief description of the 68000 instruction set.

Included with each are the addrssing mode combinations available with each opcode. Their syntax are as follows:

Dn	data register
An	address register
Dy,Dx	data registers source & destination
Rx,Ry	register source & destination (data & address registers)
<ea>	effective address - a subset of addressing modes
#<data>	numeric constant

Special notes:

The address register operands ADDA, CMPA, MOVEA and SUBA are only word and long word data sizes. The last 'A' of the operand name is optional as it is with the immediate operands ADDI, CMPI, MOVEI, SUBI, ORI, EORI and ANDI.

The ADDQ and SUBQ are quick forms of their immediate cousins. The immediate data range is 1 to 8.

The MOVEQ instruction has a data range of -128 to 127, the data is sign extended to 32 bits, and long is the only data size available.

The <ea> denotes an effective address, not all addressing modes are available with each effective address form of the instruction, as a rule program counter relative addressing is only available for the sourse operand and not the destination.

The Blitz2 compiler will signal any illegal forms of the instruction during the compile stage.

ABCD Add with extend using Binary Coded Decimal

```
ABCD Dy,Dx
ABCD -(Ay),-(Ax)
```

Data Size: byte

ADD Add binary

```
ADD <ea>,Dn
ADD Dn,<ea>
ADDA <ea>,An
ADDI #<data>,<ea>
ADDQ #<data>,<ea>
```

Data Size: byte, word & long

ADDX Add with Extend

```
ADDX Dy,Dx
ADDX -(Ay),-(Ax)
```

Data Size: byte word & long

AND AND logical

```
AND <ea>,Dn
AND Dn,<ea>
ANDI #<data>,<ea>
```

Data Size: byte word & long

ASL Arithmetic Shift Left

```
ASL Dx,Dy
ASL #<data>,Dy
ASL <ea>
```

Data Size: byte word & long

ASR Arithmetic Shift Right

```
ASR Dx,Dy
ASR #<data>,Dy
ASR <ea>
```

Data Size: byte word & long

Bcc Branch Conditionally

```
Bcd <label>
```

Data Size: byte & word

BCHG Test a Bit & Change

```
BCHG Dn,<ea>
BCHG #<data>,<ea>
```

Data Size: byte & long

BCLR Test a Bit & Clear

```
BCLR Dn,<ea>
BCLR #<data>,<ea>
```

Data Size: byte & long

BRA Branch Always

```
BRA <label>
```

Data Size: byte & word

BSET Test a Bit & Set

```
BSET Dn,<ea>
BSET #<data>,<ea>
```

Data Size: byte & long

BTST Test a Bit

BTST Dn,<ea>
BTST #<data>,<ea>

Data Size: byte & long

CHK Check Register Against Bounds and TRAP

CHK <ea>,Dn

Data Size: word

CLR Clear an Operand

CLR <ea>

Data Size: byte word & long

CMP Compare

CMP <ea>,Dn
CMPA <ea>,An
CMPI #<data>,<ea>

Data Size: byte word & long

CMPM Compare Memory

CMPM (Ay)+,(Ax)+

Data Size: byte word & long

DBcc Test Condition, Decrement, and Branch

DBcc Dn,<label>

Data Size: word

DIVS Signed Divide

DIVS <ea>,Dn Data

Size: word

DIVU Unsigned Divide

DIVU <ea>,Dn

Data Size: word

EOR Exclusive OR Logical

EOR Dn,<ea>
EORI #<data>,<ea>

Data Size: byte word & long

EXG Exchange Registers

EXG Rx,Ry

Data Size: long

EXT Sign Extend

EXT Dn Data

Size: word & long

ILLEGAL Illegal Instruction

ILLEGAL

Data Size: none

JMP Jump

JMP <ea>

Data Size: long

JSR Jump to Subroutine

JSR <ea>

Data Size: long

LEA Load Effective Address

LEA <ea>,An

Data Size: long

LINK Link and Allocate

LINK An,#<displacement>

Data Size: word

LSL Logical Shift Left

LSL Dx,Dy
LSL #<data>,Dy
LSL <ea>

Data Size: byte word & long

LSR Logical Shift Right

LSR Dx,Dy
LSR #<data>,Dy
LSR <ea>

Data Size: byte word & long

MOVE Move Data from Source to Destination

MOVE <ea>,<ea>
MOVEA <ea>,An
MOVEQ #<data>,Dn

Data Size: byte word & long

MOVEM Move Multiple Registers

MOVEM <register list>,<ea>
MOVEM <ea>,<register list>

Data Size: word & long

MOVEP Move Peripheral

MOVEP Dx,d(Ay)
MOVEP d(Ay),Dx

Data Size: word & long

MULS Signed Multiple

MULS <ea>,Dn

Data Size: word

MULU Unsigned Multiple

MULU <ea>,Dn

Data Size: word

NBCD Negate Decimal with Extend

NBCD <ea>

Data Size: byte

NEG Negate

NEG <ea>

Data Size: byte word & long

NEGX Negate with Extend

NEGX <ea>

Data Size: byte word & long

NOP No Operation

NOP

Data Size: none

NOT Logical Complement

NOT <ea>

Data Size: byte word & long

OR Inclusice OR Logical

OR <ea>,Dn
OR Dn,<ea>
ORI #<data>,<ea>

Data Size: byte word & long

PEA Push Effective Address

PEA <ea>

Data Size: long

RESET Reset External Device

RESET

Data Size: none

ROL Rotate Left (without Extend)

ROL Dx,Dy
ROL #<data>,Dn
ROL <ea>

Data Size: byte word & long

ROR Rotate Right (without Extend)

ROR Dx,Dy
ROR #<data>,Dn
ROR <ea>

Data Size: byte word & long

ROXL Rotate Left with Extend

ROXL Dx,Dy
ROXL #<data>,Dn
ROXL <ea>

Data Size: byte word & long

ROXR Rotate Right with Extend

ROXR Dx,Dy
ROXR #<data>,Dn
ROXR <ea>

Data Size: byte word & long

RTE Return from Exception

RTE Data

Size: None

RTR Return and Restore Condition Codes

RTR

Data Size: None

RTS Return from Subroutine

RTS

Data Size: None

SBCD Subtract Decimal with Extend

SBCD Dy,Dx
SBCD -(Ay),-(Ax)

Data Size: byte

Scc Set according to Condition

Scc <ea>

Data Size: byte

STOP Load Status Register and Stop

STOP #xxx

Data Size: None

SUB Subtract Binary

SUB <ea>,Dn
SUB Dn,<ea>
SUBA <ea>,An
SUBI #<data>,<ea>
SUBQ #<data>,<ea>

Data Size: byte word & long

SUBX Subtract with Extend

SUBX Dy,Dx
SUBX -(Ay),-(Ax)

Data Size: byte word & long

SWAP Swap Register Halves

SWAP Dn

Data Size: long

TAS Test & Set an Operand

TAS <ea>

Data Size: byte

TRAP Trap

TRAP #<vector>

Data Size: None

TRAPV Trap an Overflow

TRAPV

Data Size: None

TST Test an Operand

TST <ea>

Data Size: byte word & long

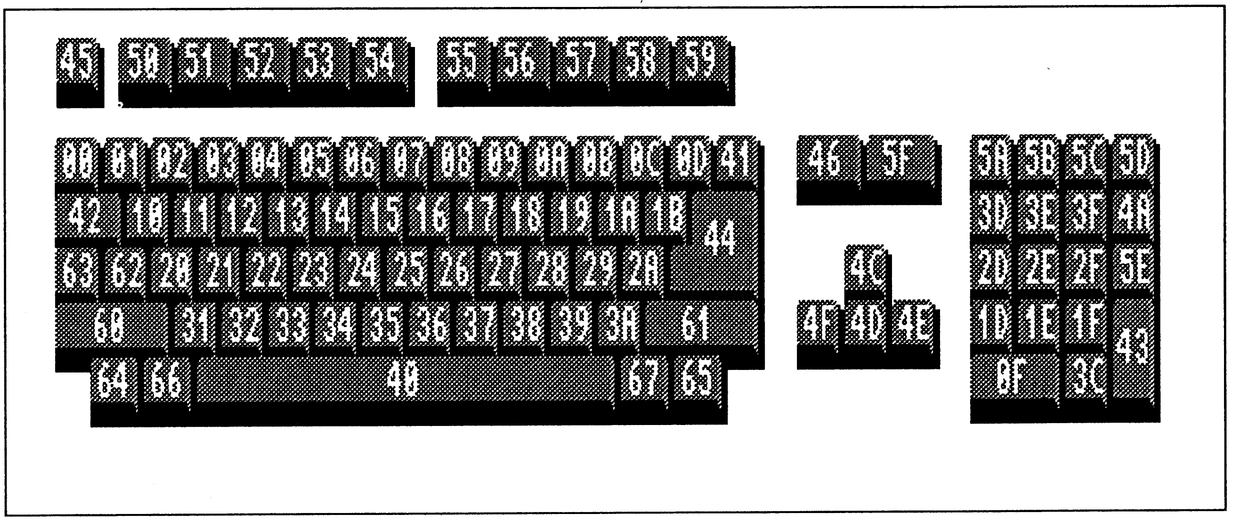
UNLK Unlink

UNLK An Data

Size: None

Raw Key Codes

The following is a diagram of all the keycodes for the Amiga 500 and 2000 keyboard. They are all in hexadecimal notation.



COMMAND INDEX