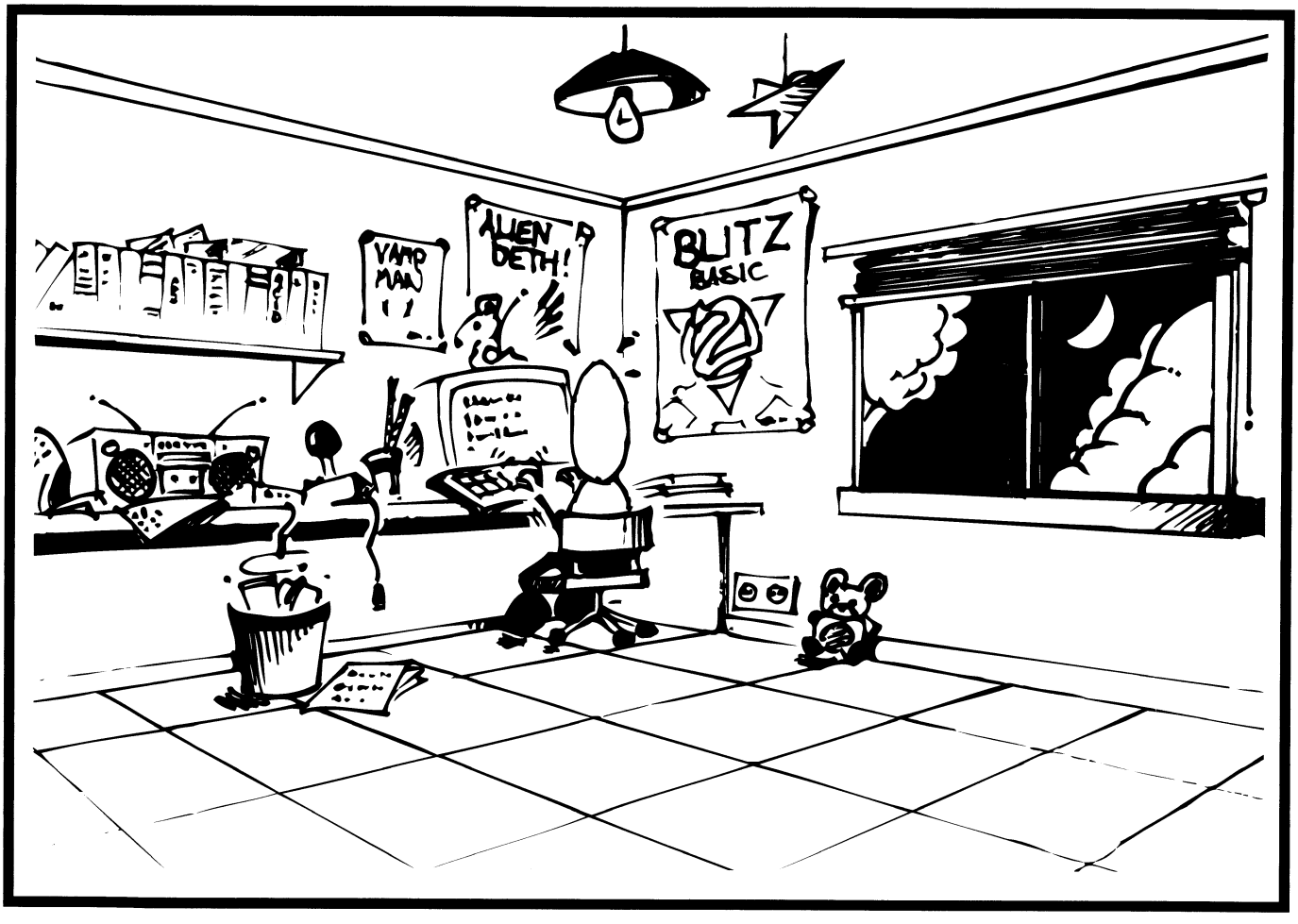
BLITZ 2 BASIC



Blitz Tech Support
1121 Alrita Ct. #4
Madison, WI 53713

Voice (608) 257-9057 9am-5pm CST
BBS (608) 257-9057 6pm-6am CST

ACID SOFTWARE

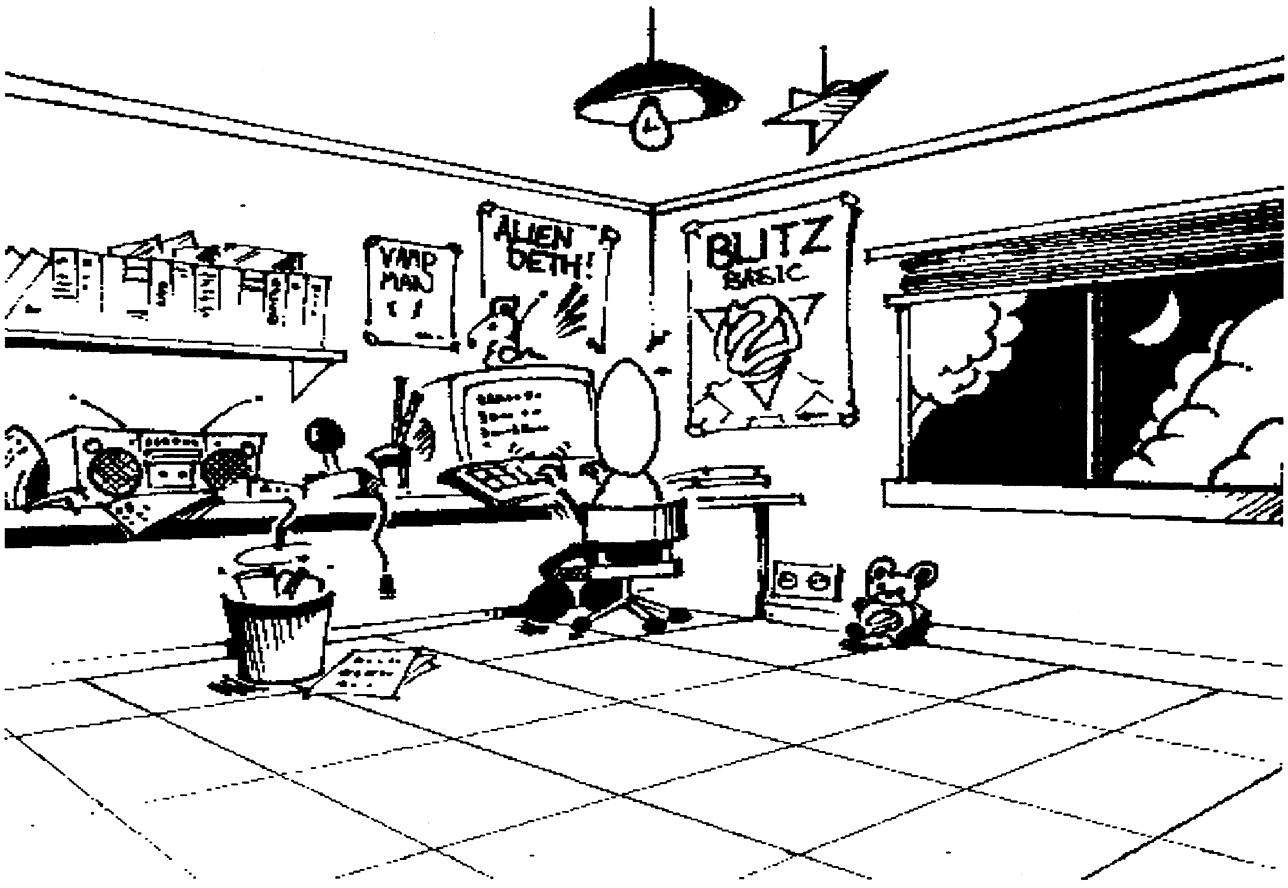
USER GUIDE

CONTENTS

Blitz BASIC 2 was developed by Mark Sibly

COPYRIGHT

Printed in Auckland, New Zealand by UNIPRINT



Credits:

Mark Sibly the author of the Blitz BASIC 2 compiler, accompanying libraries and docs.

Rod Smith for all artwork, creation of Blitz Man and the graphics on DemoDisk2.

Simon Armstrong for editing documentation and 2D routines.

Paul Andrews for writing the games on Demo Disk 2.

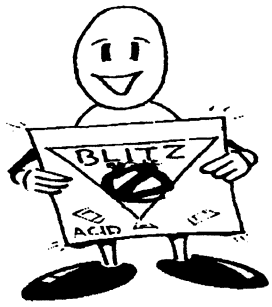
Roger Lockerbie for all the typesetting.

Rich Parrill for faithfully representing US users.

Aaron Koolen for various contributions.

Blitz One Users for all their feedback and support.

1. Getting Started



Introduction

Thankyou for purchasing Blitz BASIC 2. 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. To all of those that supported the first Blitz BASIC we say a very big HELLO! Although not a huge marketing success it gave us the confidence we needed to develop Blitz 2, which as you will find is a completely new ball game regarding power, scale, and overall expandability. However the speed remains!

The major features of Blitz 2 are its NewTypes, structures we have stolen from C and installed in BASIC where they seem quite at home. Procedures and functions are both supported for those structured programmers out there. The extended family of program control structures that BASIC has inherited from Pascal over recent years have also been supported.

LInked lists seem to be the be all and end all of programming these days and we've implemented them in Blitz 2. And whats more they're quicker than normal arrays!

We've put a lot of work into fully supporting the Amiga's operating system in Blitz 2. Menus, windows, gadgets and screens have all become Blitz 'objects' for super easy configuration and control. Then of course we've supported the Amiga library structure so any OS calls you need to make are just BASIC tokens with none of the fuss needed by other languages.

And for the speed demons, there's new methods of blitting in Blitz mode, with buffer blits that restore background graphics data when moved, as well as stencil blits that enable you to have as many depths of priority as you need.

Missing hundreds of other new features we'll just mention the Blitz 2 libraries. With the Blitz 2 built in assembler and the details found later in this manual concerning libraries, we've enabled the advanced user complete access to the compiler. Commands can be added with ease to the Blitz 2 environment and there is a huge range of hooks and compiler control logic allowing your subroutines free access to all of Blitz 2's awesome features.

Registered users of Blitz 2 can expect more indepth documentation concerning these features as well as new libraries that will continue to expand the Blitz 2 command set while continuing our reputation for sheer speed.

What You Should Have Received

1. The Blitz 2 Program Disk

This disk contains the Blitz 2 compiler and editor and various other useful files. For single floppy users the program disk can be booted fromas it contains a limited Workbench 1.3 environment.

2. The Blitz 2 Demos and Examples Disk

This disk is packed full of demos and examples written in Blitz 2.

3. The Blitz 2 Libraries Disk

This disk contains all the Blitz 2 libraries and a library manager so new libraries can be added to the Blitz 2 environment.

4. The Blitz 2 User Guide

The user guide is meant as a general introduction to programming in Blitz 2 as well as covering a wide range of specific Amiga based topics.

5. The Blitz 2 Reference Guide

The reference guide contains detailed descriptions and brief examples of the entire Blitz 2 command set as well as appendices covering compile time errors, Amiga hardware addresses and more.

A 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 Blitz 2 user.

What you should do with what you have received...

First and foremost you should back up the disks you have received and put the masters in a safe place. The simplest way to backup the disks is to boot from the program disk, select the Blitz2 disk icon by clicking on it once and then select duplicate from the workbench menus.

Special attention must be payed to the name of the disk where the Blitz2 program is located. Floppy disk users should make sure the main program disk is named "Blitz2:". The copy of Blitz that you make using workbench has to be named "Blitz2" **not** "copy of Blitz2". If it is, click on the disk icon of "copy of Blitz2", select rename from the workbench menu and delete the "copy of " part of the disk name (there should be **no** spaces in the diskname).

For HardDisk users create a new drawer from workbench and copy the files from the 3 Blitz BASIC disks into it dragging the icons with the mouse. An assign needs to be put in your startup-sequence that assigns the logical device name "Blitz2:" to the directory (drawer name) you have copied the main Blitz2 compiler and editor into.

To add the assign, load the file s:startup-sequence into a text editor such as ted, find the section where other assigns are and add a new line that reads:

assign Blitz2: dh0:blitz

The dh0:blitz is the directory name where you copied your Blitz2 program file to.

Up and Running?

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 Blitz 2 icon to run the editor/compiler.

The following are a few reasons why Blitz 2 may not start up correctly:

PLEASE INSERT VOLUME Blitz2: IN ANY DRIVE

If the above message appears you will need to quit from Blitz 2 and do the following:

HardDIsk users need to add an **ASSIGN BLITZ2: DH0:BlitzDir** line to your s:startup-sequence file and reboot your machine as explained on the previous page.

Floppy users should rename the floppy disk the Blitz2 program is on to Blitz2 as explained in the previous section. Ensure there are no leading or trailing spaces in the diskname.

NOTHING HAPPENS!

If when you double click on the Blitz2 icon from workbench and the program does not run but instead returns back to workbench it is because the editor TED is not present in the same directory as the Blitz2 program.

PLEASE INSERT VOLUME BlitzLibs: IN ANY DRIVE

If the above message appears it is because the DefLibs file is not present in the same directory as Blitz2 and TED. DefLibs contains a crunched version of all the libraries, if it is not present Blitz 2 will attempt to read all the libraries from the BlitzLibs: volume. This form of the libraries is found on the third disk you get in the Blitz 2 package which is the BlitzLibs volume that Blitz 2 is referring to.

Okee Dokee?

If you have got this far without any of the problems described above you're ready to drive the speed machine (thats the Blitz 2 editor/compiler we're talking about).

Goto the load menu, insert the examples disk and load in one of the examples. Any file ending with the suffix .bb2 is a source file able to be loaded into the Blitz 2 editor/compiler. Once you have loaded a .bb2 file have a read, guess what its going to do then select compile and run from the compiler menu.

Current Directories

If you have loaded a program from a different directory than that which the load requester has defaulted to you may have needed to click on the CD button in the load requester. This button will make the directory in the path box the current directory when you run the loaded program.

This simply means that when the program you have loaded and run does any file access such as loading graphics files it will use the current directory as the default path name. By clicking on CD before we load the file we can be sure that the current directory when we run our program is the same as the directory the program was loaded from.

Conclusion

To briefly review the points covered in this chapter, Thank you for purchasing Blitz 2, before you do anything backup the three disks you received in this package, also please don't give a copy to your friends.

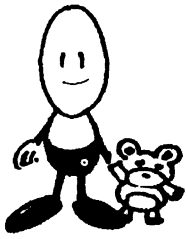
If you are a hard disk user create a drawer to hold Blitz 2 and copy all the files and drawers from the three disks into this new drawer. Then add a line to your s:startup-sequence that reads

ASSIGN BLITZ2: DH0:BlitzDrawerName

Making sure the three files Blitz2, Ted and DefLibs are in the same directory, doubleclick on the Blitz 2 icon. Select Load from the menus, then find the demos drawer on Disk 2. Select any of the files that end with .BB2 which signifies they are Blitz 2 source files (program files).

Clicking on the CD gadget before you select OK will ensure that the demo you compile&run will be able to load any files it requires. The CD gadget changes the current directory of Blitz 2 to that which the file requested is loaded from.

2. Using Ted



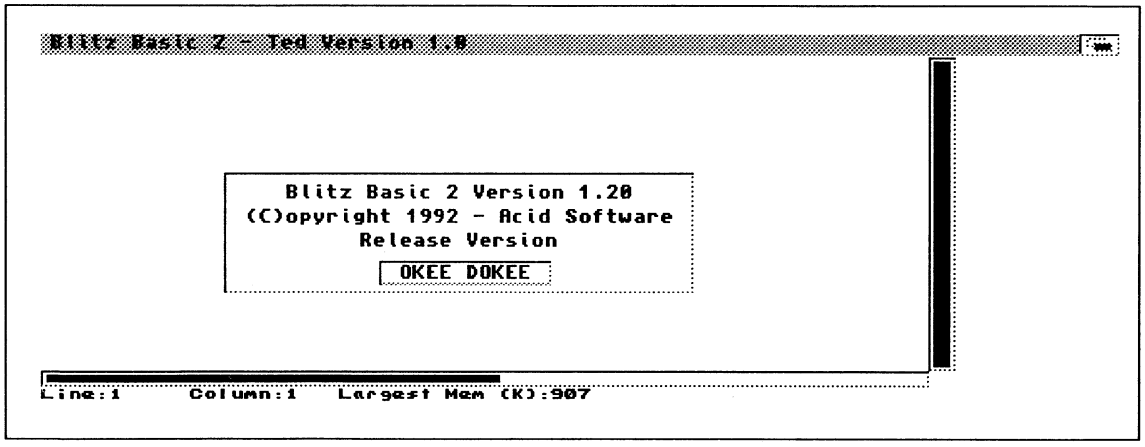
Introduction

To enter and compile your programs you need an editor. Blitz 2 comes with a text editor that acts both as an interface to the Blitz 2 compiler as well as a standalone ascii editor (ascii is the computer standard for normal text).

The following is a discussion of using ted in its ascii mode, either click on the 'ted' icon in your Blitz 2 drawer or type 'ted' from the CLI.

CLI users note: if you appreciate the user friendliness of Blitz 2's ted its a good idea to copy it into your c: directory so you can run it from any directory.

Once you have the Blitz 2 editor up and running the following screen should appear on your monitor.



The horizontal and vertical bars are called 'scroll bars', when the file you are editing is longer or wider than one page 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 Blitz 2 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.

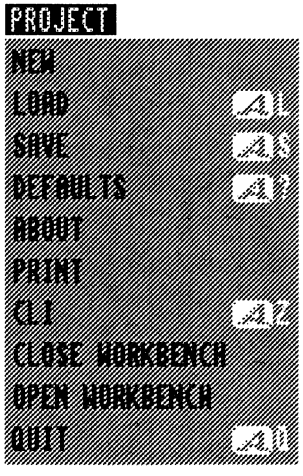
Using the shift keys in combination with the arrow keys you can move the cursor to the very start or end of a line and up and down a whole page of the document.

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 Blitz 2 editor.

The Blitz 2 Editor Menus

Using the right mouse button you can access the menu system of the Blitz 2 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 write 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 Blitz 2 editor. You can edit each of the 4 colors used, select the size of font and tell the system if you wish icons to be created when your files is saved. The scroll margins dictate how far from the edge of the screen your cursor needs to be before Blitz scrolls your view of the file. The tab setting relates to how many characters the the cursor jumps when you use the tab key. Defaults are saved in the file l:BlitzEditor.opts.

ABOUT displays version number and credits concerning Blitz 2.

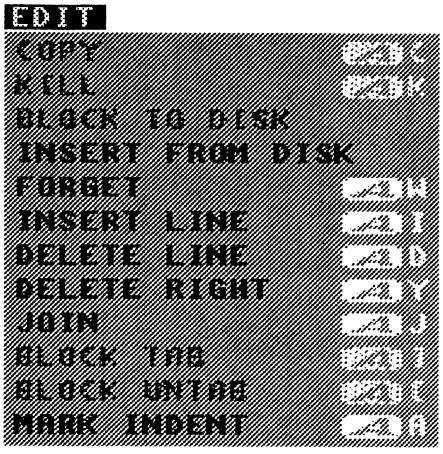
PRINT sends your file to an ouput device usually PRT: the printer device.

CLI launches a command line interface from the editor, use the ENDCLI command to close this CLI and return to the Blitz 2 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 Blitz 2 editor and returns you to workbench or CLI.

The EDIT Menu



COPY copies a block of text that is highlighted with the mouse or f1-f2 key combination to the current cursor position. The F4 key is another keyboard shortcut for COPY.

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

BLOCK TO DISK saves a highlighted block of text to disk.

INSERT FROM DISK 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.

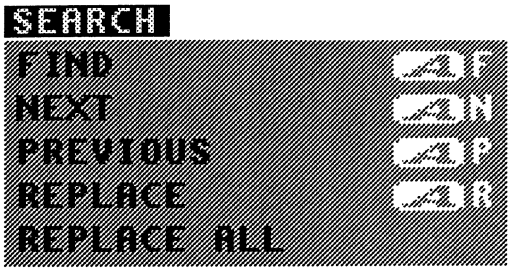
TOP moves the cursor to the top of the file.

BOTTOM moves the cursor to the last line of the file.

GOTO LINE moves the cursor to the line number of your choice.

The SEARCH Menu.

The Blitz 2 editor is able to search through your file for occurences of certain 'strings' of text such as the word HELLO.



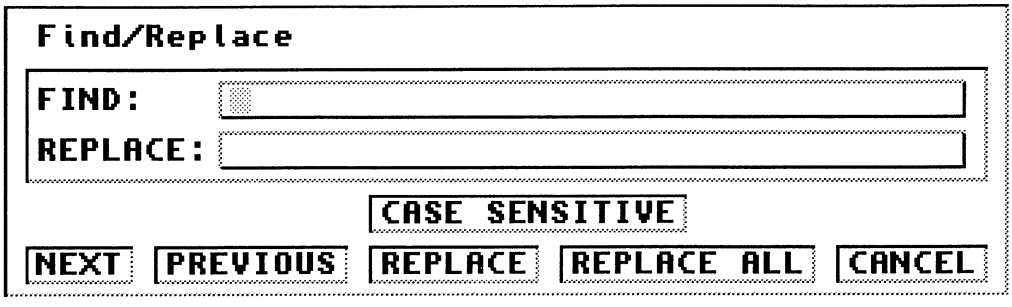
FIND will search the file for a string of characters.

NEXT will position 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.

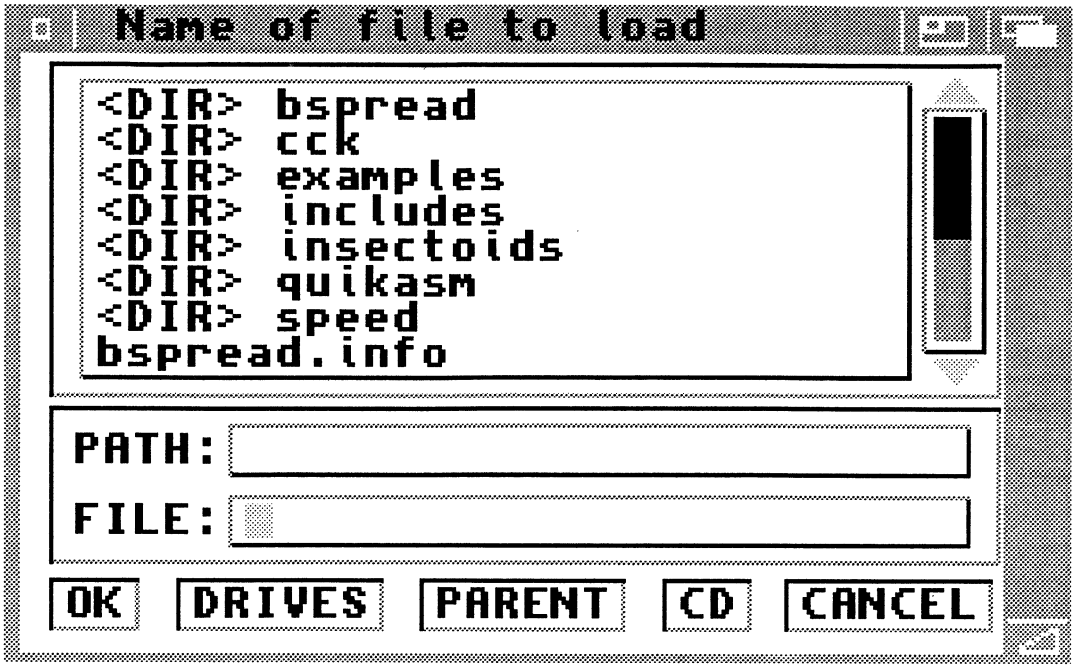
The CASE SENSITIVE option will only find strings that have the same lettters capitalised, default is that the search will ignore whether the letter are in caps or not.

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 2 FILE REQUESTER

When you select load or save Blitz 2 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 there 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 Blitz 2 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 Blitz 2 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.

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.

Keyboard Shortcuts

Having to reach for the mouse to execute some of the editor commands can be a nuisance. The following is a list of keyboard shortcuts that execute the same options that are 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 structure programming housekeeping)

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

A **D**- DELETE LINE removes line of text that 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 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 Blitz 2 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 change the look and feel of the Blitz 2 editor

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

A [- BLOCK UNTAB moves whole block left one tab

Using the Blitz 2 Editor For Blitz Programming

The following is a discussion of the extra options and commands available in the Blitz 2 editor when used in Blitz 2 programming mode. All the commands discussed in the previous chapter are available when programming with Blitz 2.

Mouseable Labels

The first thing to notice when in programming mode is the location of the vertical slide bar. This has been moved to the left to make room for 'mouseable labels'. These are labels in your Blitz 2 program that are preceded by a full stop. As you add these labels to your Blitz 2 program they automatically appear in a list at the right of the screen.

Using the mouse, click on one of these labels listed at the right. The cursor automatically jumps to the location of your program where the label is located. By labeling important routines in your program with these labels you can move quickly and easily around your code.

The COMPILER Menu

The Compiler menu includes all the commands needed to control the Blitz 2 compiler.

COMPILE/RUN will compile your Blitz 2 program to memory and if there are no errors run the program.

RUN will run the program if it has already been successfully compiled tomemory.

CREATE FILE will compile your Blitz 2 program to disk as an executable program.

OPTIONS lets you customise the Blitz 2 compiler, see next page for a detailed discussion of all the Blitz 2 compiling options.

CREATE RESIDENT will create a 'resident file' from the current file. A resident is a file including all constants and macro definitions as well as newtype definitions. By removing large chunks of these definitions from your code and creating a resident (pre-compiled) file a dramatic increase in compile speed can be 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 Blitz 2

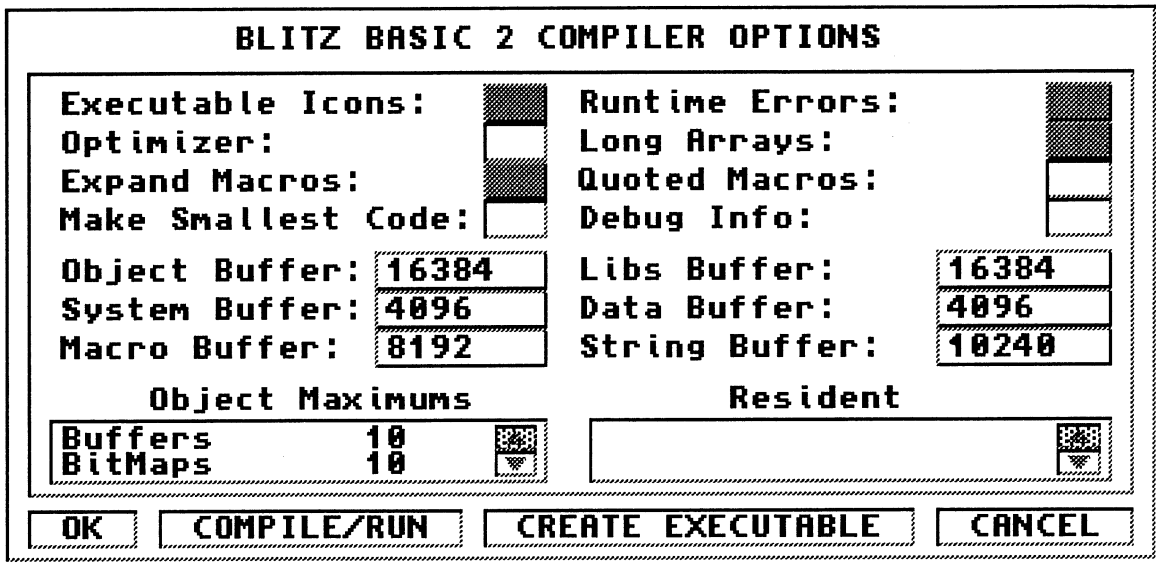
editor environment just as if you had run the program from the CLI.

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 Blitz 2 compiler environment. This is useful when writing your own Blitz 2 libraries and wish to test them without having to re-run Blitz 2.

The Blitz 2 Compiler Options

The following is a discussion of the Options requester found in the Compiler menu. A more detailed discussion of many of the features of this requester can be found in the programming chapter of this manual.



Executable Icons: 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 used at the beginning of your program.

Optimizer: switches optimizer engine on and off. Files in the Blitz2: volume that begin with BlitzOpts contain the relevant information for the Blitz 2 optimizer.

Expand Macros: selects macro expansion, if turned off Blitz 2 will not check for exclamation marks and hence compile slightly quicker.

Make Smallest Code: selects two pass compile mode, which always uses minimum amount for object code. Make Smallest should always be selected when creating executable files.

Runtime Errors: selects which error checking routines will be includedwith object code. Selecting RuntimeErrors without any of the suboptions on will still enable guru-trapping.

Long Arrays: allows arrays to be created that are greater than 32K in size.

Quoted Macros: switches macro expansion on and off when a !macro is found in quotes.

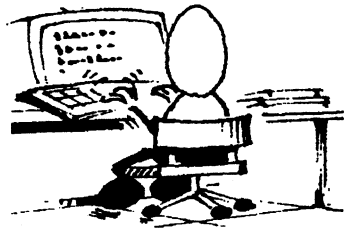
DebugInfo: creates a symbols table during CREATE FILE so executable can be debugged more easily with debuggers such as Metadigm's excellent MetaScope.

Buffer Sizes: allows different buffers to be altered when using Blitz 2 as a one pass compiler. These buffers are automatically optimised when using MakeSmallest (two pass compile).

Object Maximums: allows setting of maximum number of BlitzObjects such as screens, shapes etc.

Resident: add precompiled resident files to the Blitz 2 environment, by clicking in the box type in the resident file name, remember to add a path name if the resident is not located in the current directory (see discussion of Current Directories in the Getting Started chapter).

3. Programming



This section deals with Blitz 2's implementation of the BASIC programming language. It is essential reading for anyone wishing to take advantage of Blitz 2's many features.

Included in each section are programming examples and common mistakes relevant to the topic at hand.

1. Variable, Program-Label and Procedure Names

As with any version of BASIC, there are rules governing your choice of characters for such things as variable, procedure or program label names. Your labels MUST begin with either an alphabetic ('a'-'z') or underscore ('_') character, and may be followed by any combination of alphanumeric ('a'-'z' or '0'-'9') or underscore characters.

Label names may be of any length, and are ALWAYS case sensitive.Case sensitive means that you may have two completely different labels with the same alphabetic characters in them as long as the alphabetic characters are in a different case (ie - upper/lower case). The labels 'Name' and 'name' in Blitz 2 are two completely different labels.

In addition, labels may not begin with the name of any Blitz 2 commands.

Here are some examples of legal label names:

A

_NextPlayer

Balance

Time_Left

Player2

Here are some examples of illegal label names:

NextPlayer (starts with a BASIC command)

2Many (does not start with alphabetic or underscore character)

Column# (contains illegal character infact its a constant!)

These rules apply to ANYTHING in Blitz 2 programming which requires a label.

Program Labels

In older versions of BASIC, back in the days when 'List' was probably the most typed BASIC command, programs used line numbers to identify different sections of a program. These days, most versions of BASIC use program labels instead.

Here is an example of using a program label to identify a loop:

```
;
; A Program Label
;
main:
 a=a+1
 If a<10 Then NPrint a
 Goto main
 MouseWait
```

Here, 'main' is a used as a program label. You may have as many program labels as you wish, as long as they all have different names. In the above example, a colon ':' character was appended to the program label. This is purely for the sake of readability, and is in no way compulsory unless the label is succeeded by other code on the same line..

Summary

Blitz 2 allows labels and names to be any length.All labels and names are case sensitive in Blitz 2.
They must only start with a letter or underscore character.
They must not start with a Blitz reserved word (function or command).
Alphanumeric (a-z, A-Z, 0-9) and underscore characters only please.

2. Variables Types

The following section covers the Blitz 2 primitive types, default types as well as NewTypes, our implementation of C type structures and pointer types also a C type implementation.

Primitive Types

Blitz 2 supports 6 different types of variables. These types will be referred to through out the rest of the manual as 'primitive' types. Different primitive types are capable of storing different forms of data. For instance, the string primitive type is used to store sequences of characters, whereas the byte primitive type is used to store numbers in the range -128 to +127.

To declare a variable as being of a certain primitive type, you may follow a variable name with a 'type' descriptor. These descriptors are in the form of a dot ('.') character followed by one of the 6 primitive type descriptor characters. Here is a list of the 6 primitive types available in Blitz 2, and their descriptor characters:

Primitve	Type	Descriptor Accuracy
Byte	.b	-128 to +127
Word	.w	-32768 to +32767
Long	.l	-2147483648 to +2147483647
Quick	.q	-32768.9999 to +32767.9999
Float	.f	+/- (9*10^-18 to 9*10^18)
String	.s (or$)	<Character storage>

Bytes, Words, Longs, Quicks and Floats are all used to store numeric data. Strings are used to store sequences of characters.

The following is an example of assigning different variable types:

```
;
; an assortment of variable names and types
;
mychar.b=127
;
my_score.w=32000
;
chip.l=$dff000          ;$ denotes a hexadecimal value
;
speed3.q=500/7          ;a quick can hold fractions with 3 d.p. accuracy
;
light_speed.f=3e8       ;e denotes exponent of number 3x10^8
;
MyName$="Simon"
;
```

Once a variable has been assigned as a certain type any further reference to that variable need not use the type suffix. Beware though, the variable name may not be re-used as a different primitive type.The following will cause an error when you attempt to compile it.

```
;
;What NOT to do!
;
A.l=10      ;'A' is a long...
;
A.w=20      ;can't reuse 'A' as a word!
```

String variables are treated slightly differently. You may use either the '.s' suffix to declare a string variable, or the more traditional '$' suffix and you must include the suffix whenever you use a string variable throughout the program. As well, string variable names MAY be re-used as different primitive types.The following is quite legal:

```
;
;An example of re-using string variable names with other;primitive types
;
A.s="Hello There"      ;could also have used 'A$=...'
;
A.l=10
```

When referring to string variables, you must ALWAYS use either the'.s' or '$' suffix. Throughout the rest of this manual, we will always use the '$' suffix in examples.

Default Types

If you omit a primitive type descriptor from all references of a variable, a 'Default Type' will be used. When a program begins compiling, this default type is a Quick type ('.q'). You may alter the default type by using the DEFTYPE compiler directive. Consider this example:

```
;
;An example of using DEFTYPE
;
A=10
DEFTYPE .l
B=20
DEFTYPE .w
C=100
```

Here, 'A' will be a Quick type variable, 'B' will be a Long, and'C' will be a Word. If you are quite happy for all numeric variables in a program to be Quick type variables, you need never use suffix's or the DEFTYPE command at all.

Another form of the DEFTYPE directive involves initializing a list of variables but will **not** affect the default type for the rest of the program.

```
;
;An example of using DEFTYPE with a variable list
;
DEFTYPE .l A,B,C

D=100
```

Here, 'A', 'B' and 'C' will all be Long variables, and 'D' will be a Quick. Note how this form of the DEFTYPE directive does not actually alter the default type used for subsequent variables.

NewTypes

In addition to the 6 primitive types available in Blitz 2, programmers also have available the facility to create their own custom types, known as 'NewTypes'. NewTypes themselves must be a collection of one or more primitive types (or other NewTypes). Before a NewType can be used, it must first be defined. For this, the NewType directive is used:

```
;
; Defining a NewType
;
NEWTYPE .MyType
  A.I
  B.w
  C.q
End NEWTYPE
```

This example defines a NewType known as 'MyType'. When defining a NewType using the 'NewType' directive, you may not place any lines of normal BASIC code between the NewType and End NewType commands. Instead, you supply a list of 'entries' to be attached to the NewType (in the above example - A.I, B.w and C.q).

Once a NewType has been defined, it may be attached to any variable name in the same way primitive types are attached - via the '.' suffix.

Note that a NewType is actually just a table of separate entries. When you are referencing a NewType variable, you must specifiy the actual entry of the NewType you are refering to. For this, the backslash ('\') character is used:

```
;
;A Simple NewType
;
NEWTYPE .MyType
  X.w
  Y
End NEWTYPE

A.MyType\X=10
A\Y=20
```

This example first defines a NewType known as 'MyType'.

The line 'A.MyType\A=10' does 2 things. First, it tells Blitz 2 that the variable 'A' is of type 'MyType' (A.MyType...), and also sets the X entry of 'A.MyType' to the value 10.

The next line ('A\Y=20') sets the 'Y' entry of 'A.MyType' to the value 20. Note how in this line we did not have to specify that 'A' was of type 'MyType'. Blitz 2 already knows this thanks to the previous line. If, however, you had chosen to use 'A.MyType\Y=20', this would have worked just as well.

You may have noticed that the third line in the above example ('Y') lacked a type descriptor. Normally, this would mean that the default type would be used (usually '.q'). However, when you are defining a NewType, omitting a type descriptor from an entry name will instead use the type descriptor of the previous entry (in this case, '.w'). The first entry of a NewType MUST ALWAYS have a type descriptor.

You may also use the DEFTYPE directive with NewTypes:

```
;
;An example of a NewType
;
NEWTYPE Point3d
  X.q
  Y
  Z
End NEWTYPE
```

```
DEFTYPE.Point3d Pixel

Pixel\X=10
Pixel\Y=20
Pixel\Z=30
```

Note how both the 'Y' and 'Z' entries lack type descriptors. This means they will both use the last specified descriptor - in this case, '.q'.

The last three lines of the previous example can be written as Pixel\X=10,20,30 as Blitz2 allows multiple assigns to a NewType.

A NewType may contain entries which are themselves other NewTypes:

```
;
;An example of using NewTypes as NewType entries
;
NEWTYPE.Point3d
  X.q
  Y
  Z
End NEWTYPE

NEWTYPE.Object3d
  Coords.Point3d
  Speed.Point3d
End NEWTYPE

DEFTYPE.Object3d Ship

Ship\Coords\X=160,100,200

Ship\Speed\X=1.5,2.5,-10
```

Although the NewType 'Object3d' appears to have only 2 entries ('Coords' and 'Speed'), each entry is itself a NewType of 3 entries ('X', 'Y' and 'Z'). Therefore, the 'Object3d' NewType actually has 6 entries.

Note how the 'X', 'Y' and 'Z' entries are accessed in the 'Ship.Object3d' variable. First, the 'Coords' entry must be specified ('\Coords') and thena 'Point3d' entry ('\X', '\Y' or '\Z'). This 'nesting' of entries maybe taken to any depth:

```
;
; An example of nesting NewTypes
;
NEWTYPE.Point3d
 X.q
 Y
 Z
End NEWTYPE

NEWTYPE.Object3d
  Coords.Point3d
  Speed.Point3d
End NEWTYPE
```

```
NEWTYPE.Alien3d
  Object.Object3d
  Frame.q
End NEWTYPE

DEFTYPE.Alien3d Lander
```

Now, if we wanted to set the 'X' entry of the 'Lander.Alien3d' variable, we would use something like this:

```
Lander\Object\Coords\X=10
```

This type of referencing can be likened to AmigaDos's 'path' specification system, using the '/' character.

Note that in the above example, no type descriptors were specified in the NewType path. This is not permitted in Blitz 2, and will normally generate an error at compile time. This also applies to string entries in NewTypes:

```
;
;Strings in NewTypes
;
NEWTYPE .Customer
  Name$
  Address$
  Age.w
End NEWTYPE

A.Customer\Name="John Johnson"
```

If the last line had read 'A.Customer\Name$=...' a 'Garbage at End of Line Error' would have resulted.

The Pointer Type

The pointertype in Blitz 2 is a complex beast. When you define a variable as a pointer you also state what type it is pointing to. The following defines Biggest as a pointer to a type customer.

```
DefType *Biggest.Customer
```

The variable Biggest is just a long word that holds a memory location where some other Customer type 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 with the following code:

```
*Biggest=CustomerArray()
```

Once we have looped through the whole list we could print out the Biggest data just as if it was type Customer when it is actually only a pointer to a type customer:

```
Print *biggest\name$
```

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 are similar to C structures. By defining a NewType a variable can hold values of different types in different 'fields' known as **items** in Blitz 2. The first item in a NewType declaration must have a type suffix and any items listed without a suffix will inherit the type of the item before.

NewType **items** are accessed with the backslash \ character, similar to the way directories in AmigaDOS are accessed by the slash / character. Multiple assigns are possible by separating values by commas, each value will be assigned to the next **item** in the NewType.

A NewType may contain in it NewTypes, the items at this second level are accessed using a backslash, the inner NewType name, another backslash **then** the item name.

3. Procedures

Most versions of BASIC available today include some form of procedure support. A procedure may be thought of as a distinctly separate section of program code with it's own variable space.

Procedures in Blitz 2 may be either 'function' procedures, in which case they perform some function and return a result, or 'statement' procedures, in which case they simply perfom some function without having to return a result. Both types of procedure allow 'local' variables. Local variables are variables which are only available to the procedure, and are invisible to the main program. All local variables inside a procedure are reset to 0 each time the procedure is executed.

Procedures have much in common with normal subroutines used with the old Gosub statement, but also allow recursion (the process of a routine calling itself).

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

```
;
;An example of a procedure
;
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 pass parameters to the procedure, and in calling the procedure. These are necessary even if the procedure requires no parameters.

If you type in this program and 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.

Also note that the 'n' variable used to pass parameters to the procedure is also a local variable, and will not corrupt any global 'n's you may be using. 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 global variables (see Shared below). As well, 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.

In Blitz 2, you may also create procedures which return a value.The above example may be altered to show this:

```
;
;An example of a function type procedure
;
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}
```

The memory used by a procedure's local variables is unique not only to the actual procedure, but to each calling of the procedure. The implications of this are that a procedure may call itself without corrupting it's local variables. This allows for a phenomenon known as recursion. Here is another version of the factorial procedure which uses recursion:

```
;
;An example of a recursive procedure
;
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.

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.

```
;
;An example of using global variables inside a procedure
;
Statement example{}
  Shared k
  NPrint k
End Statement

For k=1 To 5
  example{}
Next

MouseWait
```

The Shared command tells Blitz 2 that the procedure should use the global variable 'k' instead of local '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.

A function type procedure 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'):

```
;
;An example of a function type mode that returns a string
;
Function$ spc(n)
  For k=1 To n
    a$=a$+" "
  Next
  Function Return a$
End Function

Print spc(20),"Over Here!"

MouseWait
```

Summary

Blitz 2 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 Blitz 2 function can return any primitive type using the **Function Return** commands.

4. Blitz 2 Objects

As with most versions of BASIC, Blitz 2 programs need a way to keep track of things such as multiple files or windows, often in situations where two or more of these may be 'open' or 'active' at once. Blitz 2 places special emphasis on these types of elements which are known as 'objects'. The term 'objects' in Blitz 2 can be defined as being programming elements where more than one of that element may be available at the same point in time. Therefore, things such as files, screens or windows are all Blitz 2 objects.

Creating and Destroying Objects

All Blitz 2 objects have two properties in common. Firstly, all objects must at some point be 'created'. Using files as an example, this takes place when a file is opened. Secondly, all objects must at some point be 'destroyed'. Again using files as an example, this takes place when a file is closed.

The commands used to create objects differ depending on the object you are creating. However, all commands which create objects have one thing in common - they all include an object number. This number can then be supplied to other commands which may need to know which object you are talking about. You also supply this number if you wish to destroy the object.

Here is an example of creating a file object:

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

To destroy an object, the Free command is normally used, followed by the object name and number. For example, to close file number 0, you could use:

```
Free File 0
```

In some cases, alternative commands are supplied to destroy objects. For example, a CloseFile command is also supplied to allow you to close a file:

```
CloseFile 0
```

However, Free is the most common command used to destory objects. When a program ends, any objects which have been created but not destroyed are automatically freed.

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 bitmap object. When you use the Blit command, you specify the shape object to be blitted, but not the bitmap object. If you have created more than one bitmap object, this presents the problem of how the Blit command decides which bitmap object to use. Blitz 2 achieves this through the Use command. Use is always followed by an object name and number, for example:

```
Use BitMap 1
```

After this line has been executed, any commands (for example, Blit)which require a bitmap object, but are not given a bitmap object number, will know to use bitmap 1. Bitmap 1 may be said to be the 'currently used' bitmap.

Blitz 2 makes extensive use of this 'currently used' object idea. It's advantages include faster program

execution, less complex looking commands, and greater program modularity.

Object Maximums

So, how many objects are there in Blitz 2? and what limits are there on how many of each object may be available at once? All this information can be found and altered in the options requester (found in the COMPILER menu), under the heading '**object maximums**'.

Using this requester, you can change the maximum number of any of the available objects that your program will use.

Some Blitz2 Objects

The following is a list of some of the Objects supported by Blitz 2. With the addition of new libraries more Objects will be added to the Blitz 2 environment.

Module - Used by trackerlib
BlitzFont - Used in BlitzMode for print statements
Slice - Used in BlitzMode for BitMap Display modes

Screen - Used for handling Intuition Screens
Window - Used for handling Intuition Windows

Shape - Used to hold blitter objects
Sprite - Used to hold sprite data
File - Used for I/O purposes

Palette - Used to hold colour information
BitMap - Used to hold screen images

Summary

Blitz 2's objects are custom data structures used by the libraries to handle a whole assortment of entities. Blitz 2 automatically 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 Blitz 2 commands. Parameter passing is minimised as many of the Blitz 2 commands take advantage of the currently used object.

5. Arrays and Lists

The following chapter covers arrays and linked lists, linked lists are a new concept to most BASIC programmers, delivering fast as well as very modular data handling.

ARRAYS

Arrays in Blitz 2 follow normal BASIC conventions.

All Arrays MUST be dimensioned before use, may be of any type (primitive or NewType), and may be of any number of dimensions.

All arrays begin with element 0.

The ability to use arrays of NewTypes often reduces the number of arrays a BASIC program requires. Consider this example:

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

This may also be achieved in a similar way using NewTypes:

```
;
; an array of newtype alien
;
NEWTYPE .Alien
  flags.w
  x
  y
End NEWTYPE

Dim Aliens.Alien(100)
```

You may now access all of the required alien data using just one array. If you wanted to set up all of the aliens x and y entries:

```
;
; set up random alien coordinates
;
For k=1 To 100
  Aliens(k)\x=Rnd(320),Rnd(200)
Next
```

This also makes it much easier to add more information to the aliens when required simply by adding more entries to the NewType definition.

Note: unlike most BASIC compilers, Blitz2 DOES allow the dimensioning of arrays with a variable number of elements.

LISTS

Blitz 2 also supports an advanced form of the array known as 'Lists'.

Lists are arrays, but with slightly different characteristics. For a start, you don't normally access the elements of list using an index variable. Instead, lists work on the idea of a 'current' element. Here is an example of setting up a list:

```
;
;How to prepare a list.
;
NEWTYPE.Alien
  flags.w x y
End NEWTYPE
;
Dim List Aliens.Alien(100)
```

A list starts out as 'empty', and it is up to the programmer to 'add' or 'kill' items (elements) as necessary. The above example will set up an empty 'Aliens' list, capable of storing up to 100 items.

Most list manipulation commands are in the form of BASIC functions. These functions usually return a true/false result indicating whether or not the function was successful. Continuing from the above example, here is how to 'create' 100 'Aliens' by adding elements till the list is full:

```
;
;Creating a List
;
While AddLast(Aliens())
  Aliens()\x=Rnd(320)
  Aliens()\y=Rnd(200)
Wend
```

The AddLast function adds an item to the end of a list. However,there may be no room left in a list for extra items (in the above example, there is only room for 100). Therefore AddLast will return a value representing whether or not there was any room. If an item was successfully added, a non-zero value will be returned. If there was no room, a value of 0 will be returned.

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

List processing is usually performed sequentially. To process a particular list, you normally reset the list using ResetList, then go through subsequent items using NextItem. NextItem will set the current item to the next item in the list. Here is an example of processing the list set up in the last example:

```
;
;Processing a List
;
USEPATH Aliens()

ResetList Aliens()

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

Again, NextItem returns a true/false value depending on whether there actually is a 'next item' to process.

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

```
;
;Removing items from lists...
;
ResetList Aliens()

While NextItem(Aliens())
  If Aliens()\flags=-1   ;flag for 'delete me!'
    KillItem Aliens()
  EndIf
Wend
```

After executing a KillItem, the current item is set to the one BEFORE the item removed. This means the item AFTER the one removed can be accessed via the next NextItem. To clarify this, let's assume an 'Aliens' list has been set up with 4 items in it. The words *previous of* refers to which item will be next in the list and the words *next of* refer to the item which precedes the item in the list.

item	status
1	first, and previous of 2
2	next of 1, and previous of 3
3	next of 2, and previous of 4
4	next of 3, and last

Now let's assume we process the list using **NextItem** until we get to item 2, and decide we want to remove it. However, after we have removed it, we probably want to continue processing the list from item 3 onwards. After executing a **KillItem** when we get to item 2, the list looks like this:

item	status
1	first, and previous of 3
2	(now removed! - non existant!)
3	next of 1, and previous of 4
4	next of 3, and last

As mentioned above, **KillItem** sets the current item to the one before the item removed. Therefore, after our **KillItem**, the current item will be item 1. The next **NextItem** will then set the current item to item 3, allowing us to continue processing of the list.

What happens to item 2? For a start, it can now longer be accessed by any amount of Next Iteming, as it is now longer an item in our list. Instead, it becomes 'available' to any of the item creation (eg - **AddLast**) functions.

It may also be necessary to add an item while you are processing a list, and again, you probably want to continue processing after the item has been added. The problem here is that a list may have only one current item at a time, and as soon as you add an item, the old current item will be forgotten.

There are 2 statements available for 'remembering' the current item - **PushItem** and **PopItem**. You use **PushItem** to remember the current item, and **PopItem** to later recall it.

Here is another example using the 'Aliens' list:

```
;
; example of remembering your position in a list
;
ResetList Aliens()
;
While NextItem(Aliens())
  If Aliens()\flags=-2      ;flag for 'generate a baby alien'
    PushItem Aliens()       ;remember where we were...
    If AddLast(Aliens())    ;any room? lets reproduce...
      Aliens()\x=Rnd(320)
      Aliens()\y=Rnd(100)
```

```
    EndIf
    PopItem Aliens()      ;back to where we were
  EndIf
WendResetList Aliens()
```

PushItem and **PopItem** may be nested up to 8 levels deep. Also, each **PopItem** must reference the same list as it's corresponding **PushItem**. There are many more commands available for processing Lists. They can be found in the 'List Processing' section of the Reference Guide.

Summary

Blitz 2 supports the standard BASIC array type. The **Dim** statememt is used to dimension a number of elements in an array. Blitz 2 can accept a variable number of elements and may have as many dimensions as required.

Arrays of a NewType is supported and can lead to a much cleaner more efficient programming style.

The List is a special array in Blitz 2 that is both faster and a lot tidier for arrays that need elements added and removed. Sorted arrays can also be implemented with ease with the Blitz 2's Lists.

When using lists the programmer no longer needs to keep track of the index in an array of the element they are currently using as Blitz 2 keeps track of the position in the list array.

6. Amiga mode and Blitz mode

Although the Amiga's operating system is very powerful, it can also sometimes be limiting. Certain 'tricks' at which the Amiga is very proficient at are difficult to achieve using Intuition. Therefore, Blitz 2 supports a special 'Blitz' mode of operation which can disables the operating system and allow access to some of the more esoteric things the Amiga is capable of. Using Blitz mode allows you to do such things as:

* Smooth Scrolling

* Double Buffering

* Dual-Playfield displays

* Faster program execution

However, to achieve these ends, Blitz mode does the following:

* Disables multi-tasking and Interupts

* Disables Intuition

* Disables all I/O - disk etc.

As you may have guessed, Blitz mode was mainly included for the purpose of... GAMES PROGRAMMING!

Blitz 2 programs normally run in 'Amiga' mode. This is the default mode of execution, and allows the normal access to the Amiga's operating system environment. However, through the use of the Blitz command, your programs can go into Blitz mode:

```
;
; An example of going into Blitz mode
;
BLITZ
;
MouseWait
```

If you compile and run this program, you will see that it actually does very little. However, notice that until you click the mouse button to exit, the mouse pointer has frozen! This is a normal side effect of using Blitz mode. How about a more impressive example:

```
;
; Pretty colours example
;
BLITZ     ;Go into Blitz mode

BitMap 0,320,200,1
Slice 0,44,320,200,$fff8,1,8,32,320,320
Show 0

For k=0 To 15
  ColSplit 0,k,k,k,k*13
Next

MouseWait
```

This example should give you some idea of the what you can achieve using Blitz 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!!!!!

Entering Blitz mode is a fairly rude thing to do to the operating system - Exec is completely disabled!

Therefore, 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 contains a Blitz statement.

* Hard drive users - always wait for the second blink of the drive light with 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.

The safest way to go is to ALWAYS place a VWait 250 before entering Blitz mode.

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

Blitz 2 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 Blitz 2. 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 with software dedicated to entertainment, any applications software that uses Blitz mode will NOT be welcomed by the Amiga public.

7. Strange Characters

This section deals with some of the inner workings of Blitz 2, and is therefore recommended for more advanced users.

This section is called 'strange characters' because it deals with the '&', '?' and '*' characters.

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
```

This feature is really only of use to assembly language programmers.

Finally, the '*' character can be used to create 'C' like pointers. By prefixing a variable name with a '*', you can create a 'pointer' to a variable. Consider this example:

```
;
; A Pointer Example
;
NEWTYPE .Test
  X.I
End NEWTYPE

DEFTYPE.Test A,*B

*B=A

A\X=10

NPrint *B\X

MouseWait
```

The line '*B=A' actually creates the pointer. Notice that when we assign 10 to the 'X' entry of the 'A' variable, we are modifying the output of the 'NPrint *B\X' line. This is because '*B' is 'pointing' to the data contained in the 'A' variable, so by setting up the '*B' pointer, we can read or modify anything held in 'A'.

Pointers are useful for indirect access of variables in a list. The same data can be accessed by a pointer without altering a list's current position.

Also note that pointers are completely separate from variables with the same name, but without the '*' prefix. For example, '*B' and 'B' are two completely variables and are allowed to co-exist in Blitz 2.

8. Constants

A 'constant', in BASIC programming terms, is a value which does not change throughout the execution of a program. For example, the '5' in 'a=5' is a constant. Quite often, a certain constant may be used repetitively in different areas of source code. For example, you may be using '320' as a constant to refer to the width of a low resolution screen. You may also wish to open a window on that screen of the same width - '320'. Now, there may come a time during program development when you decide that 640 would be a much better width to work with, at which point you must go through your program and change all '320's to '640's.

Blitz 2 allows you to set up special 'constants' in your programs that you may then insert into your source code instead of straight numeric characters. This allows you to easily change all occurances of a constant simply by changing the value of the constant itself. This may sound suspiciously similar to using normal variables, however there are some important differences.

*Using constants is more efficient than using variables -they don't require any memory at runtime, where as all variables do.

*Constants are defined during compiling, and when used in your program, will produce exactly the same object code as if you had used 'normal' numeric constants.

*Constants are also especially useful in combination with 'conditional compiling' (which will be covered later), when it comes to debugging programs.

Here is a simple example of constants:

```
;
;A simple example of constants
;
#Width=320
x=#Width-16
```

As you can see, using constants is very similar to using variables.The only difference is the hash ('#') character prefix used when referring to a constant. If you were to use the '#Width' constant in your program wherever relevant, it would be a simple job to later alter the value of '#Width' when fine tuning or changing your program.

Constants are of great use when defining things such as maximumlimits of arrays. Constants may be of any integer value from -2147483648 to +2147483647, and must not contain fractional values.

9. Conditional Compiling

Sometimes when writing a program, it may be convenient to have certain sections of the code temporarily ignored. This is of use when the dreaded 'debugging' stage of development occurs. A favoured BASIC method in the past has been to 'Remark out', or disable, sections of code to try to track down which part is causing a problem. Blitz 2 supplies a much more elegant method of achieving this known as 'conditional compiling'.

Conditional compiling, in combination with constants, may also be used to selectively disable and even enable sections of code based on the value of a constant. This can allow you to work on several versions of a program at the same time - for example, a 512K version and a 1 Meg version.

Here is a simple example of conditional compiling:

```
;
;This program will do nothing!
;
CNIF 1=0
  NPrint "You'll never see me!"
CEND

MouseWait
```

If you compile and run this program, it will do absolutely nothing! When Blitz 2 is compiling a program, it maintains an internal on/off flag determining whether or not object code should be produced. This 'compile flag' is normally 'on', but may be altered by the actual program being compiled.

The CNIF compiler directive in the first line of the example program represents 'conditional numeric if'. When Blitz 2 reaches a CNIF, it compares the supplied arguements and sets the compile flag to 'on' if the comparison proves to be true, or 'off' if the comparison proves to be false. CNIF comparisons may be based on any of the normal BASIC comparison characters - '=', '<', '>', '<=', '>=' or '<>'.

The CEND (conditional end) directive returns the compile flag tothe state it was in at the time the corresponding CNIF directive was encountered.

In the example program, the comparison obviously proves to be false (1 does not equal 0), therefore the compile flag will be turned off. The line between the CNIF and the CEND will never even get compiled! Once the CEND is reached, the compile flag will be turned back on.

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 Blitz 2 must be able to evaluate the comparison when it is actually compiling, and variables are not determined until a program is actually run.

Conditional compiling can be a very useful tool with which to debug your programs. When a program is not working as expected, it is often useful to insert bits of program code to test out sections - for example, to output the value of a variable you suspect may be causing a problem.

Here is an example of using conditional compiling to generate extra code to output the value of a variable:

```
;
;Debugging example
;
#debugit=1

For k=1 To 10
  CNIF #debugit=1
  NPrint k
  CEND
Next

MouseWait
```

If you compile and run this example, you will see that it prints the value of 'k' through the For...Next loop. However, if you were to set '#debugit' to 0 instead of 1, the line with the 'NPrint k' will never get compiled - therefore the For...Next will simply loop without displaying the value of 'k'.

The '#debugit' constant could be used at strategic points throughout a program allowing easy enabling or disabling of test sections.

Blitz 2 also supplies a CELSE directive similar to BASIC's' If...Else...Endif':

```
;
;A 'CELSE' example
;
#version=1

CNIF #version=1
  NPrint "Version is 1"
CELSE
  NPrint "Version is not 1"
CEND

MouseWait
```

Blitz 2 will compile only 1 of the NPrint statements, depending onthe value of the constant '#version' when compiling. CELSE actually just toggles the compile flag on or off, and may be used on it's own without the presence of a CNIF:

```
;
;CELSE and nothing else...
;
NPrint "I was here"

CELSE

NPrint "But now I'm gone!"

CELSE

MouseWait
```

In this example, the first CELSE will toggle the compile flag from on to off, therefore the second NPrint statement will never be compiled. The second CELSE will toggle the compile flag back on, allowing the rest of the program to compile as normal.

This is an ideal alternative to 'Remarking out' sections of code when debugging.

As well as CNIF, Blitz 2 gives you CSIF - conditional string if. CSIF works with literal, quote enclosed strings, and is used mostly in combination with macros...

10. Macros

Macros are a feature usually found in Assemblers or lower level programming languages, but not often in BASIC. Macros are most useful in reducing the amount of typing involved in entering repetitive sections of source code. Using macros involves first the creation of a macro definition, then one or more callings of that macro. Here is an example of a simple macro:

```
;
;A simple macro
;
Macro Inca
  a=a+1
End Macro

!Inca
```

This simple example defines the macro 'Inca' as 'a=a+1'. Now, anylines in your source code with the characters '!Inca' will behave exactly as if you had entered 'a=a+1' where you had actually entered '!Inca'. The exclamation mark ('!') tells Blitz 2 that a macro name is coming up.

Macros must first be defined before being called with the '!' character.

Macro definitions may also make use of macro parameters. This allows you to supply parameters to a macro when calling it via '!'. These parameters are supplied to the macro definitions via the back apostrophe ('`') character, followed by either a digit (1-9) or letter (a-z). This gives you up to 35 parameters to supply to your macros.

Here is a more useful 'Inc' macro:

```
;
;A Macro with a parameter
;
```

```
Macro Inc
  `1=`1+1
End Macro

!Inc{a}
```

This example will behave exactly the same as the previous example -it will produce 'a=a+1'. However, this macro is more useful because you supply the actual variable to be incremented whenever you call the macro. Therefore, you may use the same macro to increment variables other than 'a', for example:

```
!Inc{b}
```

Producing 'b=b+1'. The "1' in the macro definition tells Blitz 2 that the text supplied when the Macro is called should be used to replace the "1' in the macro definition. As this macro requires just 1 parameter, there is no need to use "2' in the macro definition. Here is a macro definition which does require 2 parameters:

```
;
;A 2 parameter Macro
;
Macro Vadd
  `1=`1+`2
End Macro

!Vadd{a,5}
```

This example is equivalent to 'a=a+5'. When the macro is called,the first parameter supplied in the curly parenthesis ('a') will be inserted into the macro wherever there is a "1' in the macro definition. Similarly, the second parameter ('5') will replace the "2' in the macro definition.

If you are creating a macro which uses more than 9 parameters, you can start using "a', "b'... to represent parameters 10,11 and so on. This imposes a limit of 35 (9 digits + 26 letters) parameters on your macros.

It is important to remember to supply enough parameters when calling a macro to satisfy the number of parameters in the macro definition. Leaving out parameters when calling a macro will result in 'nothing' being put in the place of the missing parameters. For example, using the '!Vadd' macro like this:

```
!Vadd{a}
```

Is equivalent to 'a=a+'. This will result in an error when the program is compiled. If you wish to write macros which can trap these errors and generate appropriate error messages, you can use the special '0 parameter in macros. The '0 parameter gets replaced with the number of parameters supplied when a macro is called. You can use '0 along with conditional compiling to check that sufficient parameters have been supplied to a macro:

```
;
;A Macro with error checking
;
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.

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 generat the right label.

```
;
; cmake example
;
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 21as Blitz 2 would expand the the code after the NPrint to read var2+1.

11. Blitz 2 Errors

Program errors in Blitz 2 come in 2 varieties - compile time errors and runtime errors.

Compile time errors are reported when a program is being compiled including such things as syntax errors, unknown labels and the like. Compile time errors are errors which prevent a program from being successfully compiled.

Runtime errors occur while a compiled program is running, and include such things as illegal function calls. Runtime errors are errors which occur once a program has been successfully compiled and is actually running. By use of the SetErr command, it is possible for your programs to 'trap' runtime errors. Here is an example of SetErr in action:

```
;
; a seterr example
;
Dim a(10)

SetErr
  NPrint "Runtime Error! Click mouse to exit..."
  MouseWait
  End
End SetErr

For k=1 To 11
```

```
  a(k)=k
  NPrint a(k)
Next
```

If you look closely at this program, you will see that the line 'a(k)=k' will eventually cause a runtime error when the variable 'k' gets to 11. When this happens, program flow will be transferred to the SetErr routine.

The ClrErr command may be used to return to normal runtime error reporting, like so:

```
;
; a clrerr example
;
Dim a(10)

SetErr
  NPrint "Runtime Error! Click mouse..."
  MouseWait
  ClrErr
End SetErr

For k=1 To 11
  a(k)=k
  NPrint a(k)
Next
```

Normally, to be able to use SetErr, 'runtime errors' must be enabled in the compiler options requester. However, there are a few situations in which this is not necessary:

* When any command which requires a filename cannot locate the specified file. For example, LoadShape, LoadBitMap.

* When an attempt to open an Intuition screen or window is unsuccessful. This may be because the parameters supplied are unusable.

* When Blitz 2 runs out of memory.

12. Resident Files

To make writing programs which manipulate large number of NewTypes, macros or constants easier, Blitz 2 includes a feature known as 'resident files'. Resident files keep you from having to re-compile sections of a program which define NewTypes, macros or constants. As their name suggests, resident files are disk-based files in a special format usable by Blitz 2. A resident file contains a pre-compiled list of NewTypes, macros and constants, and may be accessed by any Blitz 2 program. There are 2 steps involved in using resident files - creating resident files and accessing resident files.

To create a resident file you will need a program which contains all the NewTypes, macros and constants you which to convert to resident file format. Here is an example of a such a program:

```
;
; a simple resident type file
;
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 and run' the program, and 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 accessed by 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 be automatically available.

13. Operators and Constants

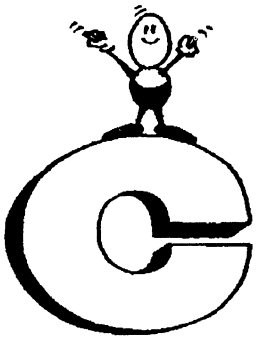
The following chart describes all operators available in the calculation of expressions in Blitz 2. Operators are listed in order of precendence, from highest to lowest. LHS and RHS stand for the left hand side and right hand side operands of the operation.

Operator	Result
NOT	RHS logically NOTted
-	RHS arithmetically negated
BITSET	LHS with RHS bit set
BITCLR	LHS with RHS bit cleared
BITCHG	LHS with RHS bit changed
BITTST	true if LHS bit of RHS is set
^	LHS to the power of RHS
LSL	LHS logically shifted left RHS times
ASL	LHS arithmetically shifted left RHS times
LSR	LHS logically shifted right RHS times
ASR	LHS arithmetically shifted right RHS times
&	LHS logically ANDed with RHS
|	LHS logically ORed with RHS
*	LHS multiplied by RHS
/	LHS divided by RHS
=	true if LHS is equal to RHS
<>	true if LHS is not equal to RHS
<	true if LHS is less than RHS
>	true if LHS is greater than RHS
<=	true if LHS is less than or equal to RHS
>=	true if LHS is greater than or equal to RHS
AND	LHS logically ANDed with RHS
OR	LHS logically ORed with RHS

Blitz 2 also supports the following constants:

Constant	Value
Pi	3.1415
On	-1
Off	0
True	-1
False	0

4. C Concepts



The following chapter covers some of the C programming concepts that have been integrated into Blitz 2. A conversion utility for C include files is listed at the end of the chapter to illustrate the differences in Blitz 2 from C programming.

Defining & Assigning Structures/NewTypes.

A structure in C is similar to a record in a database. It has a number of fields that hold different types of information. Structures in Blitz 2 are known as NewTypes and their definition is similar to C.

Original C structure:

```
struct mystruct
{
  char age;
  int  height,width;
  char  name(50);
)
mystruct myvar;
```

Blitz 2 equivalent:

```
NewType  .mystruct
  age.b
  height.l:width
  name.b(50)
End NewType

DefType .mystruct myvar
```

The first difference in the Blitz 2 version is that the type of each field is defined by the .type suffix rather than the preceding type definition in the C version.

Secondly, multiple fields of the same type are listed with commas in C. In Blitz 2 any fields listed without a .type suffix inherit the type of the previous field.

The square brackets work the same in both Blitz 2 and C as well as the ability to use NewTypes within NewTypes.

Finally, the name.b[50] field in the above example should be replaced by name$ as Blitz 2 fully supports string handling within NewTypes. However Blitz allocates a long word pointer to string memory when a string variable is listed in a NewType so only 4 bytes are allocated in the structure raising some compatibility problems.

Both of the above examples setup the variable myvar to be of type mystruct. With Blitz 2 using the suffix .mystruct with the first instance of the variable myvar would also define it's type.

Accessing fields in Structures/NewTypes

To gain access to a variable's different fields Blitz 2 uses the backslash (\) as opposed to C's use of the full stop (.) or arrow characters (->).

Blitz 2 allows multiple assigns to a variables different fields. The following example will assign values to all the fields of the variable myvar, the commas separate the list of values.

myvar\age=20,190,40,"Aaron"

Note that the above will only work when the name field is defined as type string in the NewType definition.

Arrays of Structures/NewTypes

Creating arrays of structures in Blitz 2 is similar to C. The following creates an array of 100 variables of type mystruct.

C Version

```
struct mystruct lots(100);
```

Blitz 2 version

```
Dim lots.mystruct(100);
```

Once again Blitz 2 requires the suffix .mystruct to be attached to the array name and C needs the struct name preceding the variable name.

Unions

Unions can be cludged in Blitz 2 with relative ease.

C Version

```
struct mystruct
{
union aunion
(
  char id_type;
  long id_val;
  word id_len;
}
 my union;
}
```

Blitz 2 Version

```
NewType.mystruct
  id_type.b(0)
  id_val.l(0)
  id_len.w(0)
```

```
    myunion.l
End NewType
```

Preceding a variable with [0] in Blitz 2 will allocate zero bytes of memory for that field. This means the offset of the next entry listed will be the same as the previous. This allows the programmer to configure different types in a NewType at the same offset. The [0] suffix is not required when accessing the field from your code.

Pointers

Pointers in Blitz 2 are a little different than their C counterparts.

In C the following line copies a byte from one place to another:

*pdest++=*psource++;

The same line in Blitz would need to read:

*pdest\charoffset=*psrc\charoffset:*pdest+1:*psrc+1

Firstly Blitz requires the pointer to be a NewType type for it be used to access the memory it points to. This is because Blitz treats a pointer as a long if it is not succeeded by a backward slash. If it is succeeded by a backward slash Blitz uses the pointer to locate the type in memory and generates the offset using what follows the backslash.

Thus in the previous example charoffset is an item in a NewType.

*pdest\charoffset=*psrc\charoffset copies the byte pointed to by pdest to psrc

*pdest+1 and *psrc+1 treat the pointers as standard longs and add 1 to each meaning they point to the next byte in memory

A NewType with a single entry of charoffset.b should be defined to use the above example effectively.

The above is a little confusing as pointers in Blitz 2 were not designed for string type manipulation but for indirect access of NewTypes that are buried in lists and arrays.

#Defines and Blitz 2 Constants and Macros

A Blitz 2 constant is any numeric preceded by a hash sign (#). #MyMax=10 defines the constant MyMax just as #Define MyMax 10 would do the same thing in a C program. Any use of the constant in your code should always be preceded by the hash character (#).

When text is involved with C #Defines macros need to be used in Blitz 2.

```
#define IsListEmpty(x) \
    ( ((x)->lh_TailPred) == (struct Node *)(x) )
```

would be replaced by the Blitz 2 macro:

```
Macro IsListEmpty:
  `1\lh_TailPred=`1
End Macro
```

and would be accessed in your code using

```
If IsListEmpty{*ListPtr} Then ...
```

Note the use of the reverse single quote character (above the [Tab] key) to denote the first parameter passed in the curly brackets. This example is also another illustration of the different treatment of the pointer type in Blitz 2.

Include Files

Include files are treated similarly in both C and Blitz 2. Blitz 2 has two extra commands of interest in this topic. **IncDir** defines the pathname that is used by AmigaDos to locate the filename specified after the **Include** statement. **XInclude** is the same as the Include statement except that if the file has already been included previously in the code by perhaps another Include file Blitz 2 will not insert it into the code a second time.

Conversion Program for C Header Files

The following program is designed to make the conversion of Amiga .h files to Blitz 2 simpler. It's main function is to rewrite the structure definitions as NewType definitions. It also converts #defines to Blitz 2 constants when they are numeric in nature and to macros when they are text replacement #defines.

Because of the XInclude statement in Blitz 2 it also removes all the Ifndef logic that Amiga C header files contain. Note a few other details such as << and >> being replaced by the LSR and LSL logical shift operators.

Because the authors are by no means C experts future revisions of this program are expected to be provided by C knowledgable Blitz 2 users. This may also explain any errors in this chapter but HEY! at least we made the effort.

```
;
; .h header file converter to a bb2 semi-readable file
;
; a lesson in cludge city by simon
;
; v1.0
; ----
;
; rems out /* */ sections with ;
;
; converts structs to NewTypes
;
; converts types in structs to bytes words and longs
; (no string types as yet)
;
; handles #define of constants
;
```

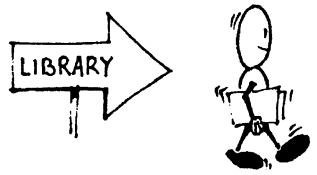
```
; converts #define "ascii" to Macro..End Macro
; (also replaces (x) parameter to `1)
;
; ignores all ifndef logic because #include => xinclude
;

Function$ getline{}
 Shared cp
 poo
 a$=Edit$(128):cp=1
 a$=Replace$(a$,";","'")
 a$=Replace$(a$,Chr$(9)," ")

 If Instr(a$,"/*")>0          ;whole comment
  If Instr(a$,"*/")>0
   a$=Replace$(a$,"/*",";/*")
  Else
   a$=Replace$(a$,"/*","**")
   NPrint ";",a$
   blockloop:
    a$=Edit$(128)
    If Instr(a$,"*/")=0
     NPrint ";",a$
     Pop If
     Goto blockloop
    End If
   a$=Replace$(a$,"*/","**")
   NPrint ";",a$
   Pop If:Pop If
   Goto poo
  End If
 End If
 Function Return a$
End Function

Function$ nextword{a$}
 Shared cp
 b$=""
 While Mid$(a$,cp,1)=" ":cp+1:Wend   ;skip leading spaces
 If Mid$(a$,cp,1)=";"
  Function Return Mid$(a$,cp)
 EndIf
 While cp<=Len(a$) AND Mid$(a$,cp,1)<>Chr$(32)
  b$+Mid$(a$,cp,1):cp+1
 Wend
 Function Return b$
End Function

If NumPars<>1
 NPrint "Wrong Number of Parameters"
 End
End If

CaseSense Off

n$=Par$(1):n$=Replace$(n$,".h","")+".bb2"
```

```
If NOT ReadFile(0,Par$(1)) Then NPrint "Couldn't Open File":End

If WriteFile(1,n$)
 FileInput 0:FileOutput 1

 While NOT Eof(0)
  ;
  a$=getline{}
  If Left$(a$,1)=";" Then NPrint a$:@$Goto dunline
  c$=nextword{a$}
  If c$="struct" Then Gosub dostruct
  If c$="#define" Then Gosub dodefine
  If c$="#include" Then Gosub doinclude
dunline:
 Wend
EndIf
End
;-------------------------------------------------------------
.doinclude:
 v$=nextword{a$}
 NPrint "XINCLUDE ",v$
 Return
;-------------------------------------------------------------
.dodefine:
 v$=nextword{a$}
 e$=nextword{a$}

 If (e$>="a" AND e$<="z") OR (e$=>"A" AND e$<="Z") ;string define
   ps=Instr(e$,"."):If ps Then e$=Mid$(e$,ps+1)
   f$=nextword{a$}
   NPrint "Macro ",v$,":",e$,":End Macro",Chr$(9),f$

 Else

 If e$="\"
  s$="":p=Instr(v$,"(")
  If p Then s$=Mid$(v$,p):v$=Left$(v$,p-1)
  NPrint "Macro ",v$      ;cludge it man
  a$=Edit$(128)
  If s$<>"" Then a$=Replace$(a$,s$,"`1")
  a$=Replace$(a$,"->","\")
  NPrint a$
  NPrint "End Macro"
 Else
  f$=nextword{a$}                    ;numeric define
  e$=Replace$(e$,"0x","$")
  e$=Replace$(e$,"L","")
  e$=Replace$(e$,"L<<","LSL")
  e$=Replace$(e$,"L>>","LSR")
  e$=Replace$(e$,"<<","LSL")
  e$=Replace$(e$,">>","LSR")
  If e$="" Then e$="-1"
  NPrint "#",v$,"=",e$,Chr$(9),f$
 EndIf

 EndIf
```

```
 Return
;-------------------------------------------------------------
.dostruct:
 NPrint "NewType ."+nextword{a$}
 u=0

nxline:
 a$=getline{}

 If a$="" Then Goto nxline
 If Left$(a$,1)=";" Then NPrint a$:@$Goto nxline

 c$=nextword{a$}

 c$=UCase$(c$):t$=""

 Select c$

 Case "}"
  If u
   u-1
  Else
   NPrint "End NewType"
   Pop Select
   Return
  EndIf

 Case "UNION"
  u=2:Pop Select:Goto nxline

  Case "UWORD":t$=".w"
  Case "WORD":t$=".w"
  Case "SHORT":t$=".w"
  Case "USHORT":t$=".w"
  Case "UNSIGNED":t$=".w"
  Case "SIGNED":t$=".w"
  Case "INT":t$=".w"        ;is this default word on the amiga?
  Case "VOID":t$=".w"
  Case "BOOL":t$=".w"

  Case "BYTE":t$=".b"
  Case "UBYTE":t$=".b"
  Case "CHAR":t$=".b"

  Case "LONG":t$=".l"
  Case "ULONG":t$=".l"
  Case "APTR":t$=".l"
  Case "BPTR":t$=".l"
  Case "CPTR":t$=".l"
  Case "STRPTR":t$=".l"
  Case "BSTR":t$=".l"
  Case "FLOAT":t$=".f"

  Case "STRUCT":t$="."+nextword{a$}

 End Select
```

```
 If t$<>""
  n$=nextword{a$}
  b=Instr(n$,"("):b$=""
  If u=2 Then b$="(0)":u-1
  If b Then b$=Mid$(n$,b):n$=Left$(n$,b-1)
  NPrint "  "+n$+t$+b$+Chr$(9)+nextword{a$}
 EndIf

Goto nxline
;-----------------------------------------------
```

5. Blitz 2 Libraries



Blitz 2 was designed to be as flexible and open-ended as possible. To achieve this, Blitz 2 is largely library driven. Blitz 2 libraries ('blitzlibs') contain the bulk of Blitz 2's command set - including token text, help text, and actual code. The following chapter covers the assorted libraries included with Blitz 2, details on converting Amiga libraries to work with Blitz 2 and details on developing your own libraries in Blitz 2.

Library Configuration

When Blitz 2 is run, it searches for a file called **DefLibs** in the Blitz2: directory. DefLIbs contains a collection of Blitz2 libraries. If DefLibs is not found Blitz 2 then loads its libraries from the BlitzLibs: directories library by library.

The LibManager program included on the Program Disk enables the user to select which of the Blitz 2 libraries are included in the DefLibs file. The DefLibs file that is included on the Program Disk includes all the libraries. Libmanager should be used when revisions and additions are made to the BlitzLibs: directories.

For development systems with limited memory DefLibs can be used to select a limited number of the Blitz 2 libraries. However object code produced by the compiler will always only include the libraries used by the commands within the program so no memory will be saved regarding compiler output.

Tokens

The Blitz 2 editor stores all the BASIC keywords as tokens, these tokens are 16 bit numbers encoding which library the BASIC keyword comes from and an offset detailing where in the library the keyword's ASCII name, help information and compile time code can be found.

Because the user can configure Blitz 2 to use some libraries and not others problems can arise when a program's source code includes tokens that are not available in the libraries that Blitz 2 loaded when it was run. The error Token Not Found $xxxx signals the token code it was unable to locate.

Each library can include 128 statements and functions. The token code uses the lower 7 bits for which function/ statement it is and the upper 9 bits to signify which library the token came from.

The SYSTEM Libs Directory

The system libs contain the core memory management, string, list and array handling libraries. The SYSTEM directory of BlitzLibs: should never be changed by the user except when upgrading licensed versions of the SYSTEM drawer from Acid Software.

The BASIC libs directory

The BASIC directory in BlitzLibs contains the main Blitz 2 command set. These libraries contain all the Blitz mode and Amiga mode type functions and statement. The directory should only be expanded with licensed libraries from Acid Software.

The AMIGA Libs Directory

The AMIGA directory of BlitzLibs: contain standard Amiga libraries, supplying token text, help text and other information necessary to the compiler. Blitz handles the loading of the library base, the parameter passing details and the return value so the user can treat any Amiga library call just like a BASIC keyword.

Blitz 2 ships with the intuition, diskfont, exec, graphics and the dos libraries in the AMIGA directory. These must not be removed as Blitz uses them for many of its own commands. Any other Amiga library can be added to the Blitz 2 environment by using the convertfd utility included on the Blitz 2 Program Disk.

The AMIGA libraries add an underscore character to all the tokens to avoid conflict with Bltz based tokens. Also each library call can either be treated as a BASIC function or a statement. An example of using calling routines in the DOS library follows.

```
;
; Amiga library calls example
;
handle.l=Open_("ram:test",1006)  ;Blitz opens the DOS library for us
                                 ;OPEN_ returns a handle
If handle                        ;if the handle is non-zero
 Write_ handle,memloc,length     ;as a function Write returns num bytes written
 Close_ handle
EndIf
```

When using many Amiga libraries the C structures and constants will also need to be accessed to properly utilise the libraries. The C Programmers chapter discusses relevant Blitz conventions and includes a program listing to convert .h header files to Blitz code. These can then be pre-compiled as residents for a fast, efficient environment.

The USER libs directory

The USER directory is for including any public domain Blitz based libraries as well as libraries you may wish to develop. An example of a simple library is included at the end of this chapter as a starting point to those users wishing to develop machine code libraries for their BASIC environment.

Creating Custom Libraries

This section is for the competent machine code programmer who wants to add libraries of commands to Blitz2.

Whole sets of Statements and Functions can be added to Blitz 2 that are integrated fully into the Blitz 2 environment including the tokenising of their names and online parameter help within the Blitz 2 editor.

Library routines are called directly and parameters are passed when possible via 68000 registers. This gives Blitz 2 a real performance advantage over other compiled languages such as C.

Design Aspects Of Blitz 2 Libraries.

A library is ideally a group of highly optimised machine code routines that provide all the necesary functions in their related field. For instance a hypercard library should cover all the primitives necesary to implement the algorithms necesary for a hypercard type environment.

If you plan on distributing your library to other users it should be registered with Acid Software for an official priority number. If you plan on marketing your library it MUST be registered with us as we need to uphold standards in association with error trapping, memory management etc.

If you want to write a Blitz 2 library but don't know what to do may we suggest a few topics that might be fun to try...

* a text parser for adventure games and the like

* the complete set of Forth Language primitives

* a machine code rewrite of intuition using the same commands

* an imaginary numbers library

* a faster 3D library than that included with Blitz 2

* image processing algorithms

* fuzzy logic routines for AI program development

As we mentioned before, each function and statement should ideally have error checking routines that will be switched on and off from the Blitz 2 compiler options.

Then there's the syntax and documentation to think about. Try and follow the rules that we have set down in the Blitz 2 Reference Manual. Don't hesitate to get in touch with us.

LibMacs and Blitz 2 Configuration Requirements

Included on the examples disk is a set of machine code macros and constants to make the task of creating your library simpler called LibMacs. LibMacs.res is the resident form of these macros, making sure it is in your current working directory call up the options requester and type libmacs.res into the residents window.

While the option screen is up, also turn off all error checking and select the make-smallest option. This is so we can use Blitz 2 as a raw assembler without it inserting header code and extra buffers at the end of the destination code (our library.obj file).

Once you have the options setup as above you are ready to write your first Blitz 2 library.

The libheader macro must be used at the top of every library.

```
!libheader {LIBRARYNUMBER,SETUPROUTINE,RETURN,CLOSEROUTINE,ERRORROUTINE}
```

A library can have its own initialisation and close down routines which are automatically called by the

Blitz 2 runtime module at the start and end of program. Your initialisation routine can return a value (usually a pointer to some global data it creates) for other libraries lower in priority to access.

High priority libraries that for instance provide a set of fast graphics primitives should return a pointer to any tables that could be used by other libraries.

A low priority library that just provides a set of commands to be accessed from Blitz 2 only needs a libnumber and would read.

```
!LibHeader {LIBRARYNUMBER,0,0,0}
```

A CloseRoutine may clear any locations that specify an allocated memory block. Because Blitz will automatically free memory allocted with the memlib calls $c002 and $c003 (see interfacing with other libraries) your library's close routine does not have to free up the memory. However if you 'RUN' your program from Blitz 2 without re-compiling it locations such as label:dc.l 0 will **not** be reset to zero. So your library may need a Close routine that resets these locations to zero.

A library can contain a maximum of 128 Statements and Functions. These will become official tokens in Blitz 2 when that library is in use so care must be taken not to duplicate token names from other libraries.

The macros Statement Function and Command are in the form..

```
!astatement {}          ;no parameters

!afunction {RETURNTYPE}  ;#byte #word #long #quick #float #address or #string

!acommand {RETURNTYPE}  ;#byte #word #long #quick #float #address or #string
```

A command form specifies that the user can treat the keyword as either a function or a statement.

Each statement and function in a library has a certain number of parameters that are passed in registers d0 to d5.

The args (arguments) macro should follow after every statement and function macro and defines the types and number of parameters that are to be passed to the routine.

```
!args{#type{,#type..}}  ;#type=#byte #word #long #quick #float #address or #string
```

The libs (libraries) macro specifies other library bases, init returns, objects etc. that need to be passed to your routine. For added control a parameter in the args macro can also be used as an index for specific object numbers etc.

The following example tells the Blitz 2 compiler to generate the location of the BitMap number using the first parameter from the args list as the BitMap number and place the result in a0.

```
!libs{#BitMapLib,#pd0+#ia0}  ;paramater d0 specifies BitMap# which is passed ina0
```

The subs macro has three parameters, the first points to the actual machine code subroutine to be called by Blitz 2 after it has configured the parameters. The second is an optional error checker routine pointer that should check parameters if the user has invoked error checking. The last parameter should be 0 for compatibility reasons.

Note your subroutine name should be different than that of the token name as it will tokenise after your library is installed!

```
!subs{_name,0,0}
```

The name macro tells Blitz 2 the name of the routine which it will use as a token and the help text that will appear when the Blitz 2 user hits the help key on the token.

```
!name{"name","helptext"(,label)}
```

After listing all the commands finish the Header with

```
!libfin
```

A Simple Blitz 2 Library

The following is the source code for a very simple library that includes the function MyPeek and the statement MyPoke.

After making an executable of your library in the the Blitzlibs:userlibs directory do a ReLoadLibs from the compiler menu. Blitz 2 should now have the commands MyPeek and MyPoke which should be highlighted when you type them in and have the Help text that we included in the second parameter of the name macro. They should hopefully also be fully functional!

```
!libheader{50,0,0,0,0}

!afunction {#word}
!args {#long}
!libs
!subs {_mypeek,0,0}
!name {"MyPeek","location"}

!astatement
!args {#long,#word}
!libs
!subs {_mypoke,0,0}
!name {"MyPoke","location,value"}

!libfin

_mypeek:MOVE.l d0,a0:MOVE (a0),d0:RTS

_mypoke:MOVE.l d0,a0:MOVE d1,(a0):RTS
```

Do not put an **End** statement in the code or any other BASIC command as Blitz 2 will put its own variable initialisation routines at the start of the object code infront of the library header.

The following two pieces of code should test the two commands we have added to our userlibs directory.

```
While Joyb(0)=0          ;while no mouse button down
 MyPoke $dff180,n:n+1    ;poke color 0 register
Wend

While Joyb(0)=0          ;while no mouse button down
 NPrint MyPeek($dff006)  ;read video beam position
Wend
```

Complex Blitz 2 Library Topics

Are you messing with a3-a6?

Your command must restore registers a3-a6 before returning. All other registers can be returned in any state of disrepair. Oops, Mark says that a3 is cool to mess with in procedures, but NOT functions.

More than 6 Parameters?

If your command has more than 6 parameters the first 6 are passed as usual in registers d0-d5 and the remaining are passed in a list pointed to by the address register a2.

To access these just:

```
move.w-(a2),parameter7
move.l-(a2),parameter8   ;(long)
```

Passing String variables?

If your command uses string parameters a pointer to the string is passed as usual and the length can be found in the table pointed to by a2.

Note: string lengths are long words and should be pulled off the table before parameters7 onwards are.

Using String Parameters in Functions?

Care must be taken with Blitz 2's string work area in functions. The register a3 points to the current position in this work area.

With numeric functions that use string parameters such as len(a$) the register a3 must be returned pointing to the beginning of the string work area. This is the same as the pointer to the first string parameter passed to the function.

Using a String Function?

If it is a string function i.e. returns a string, the returned string has to start at the beginning of the string work area, a3 should point to the end of the string and d0 should equal the length of the string.

Confused?

O.K. Lets have a quick section on strings.

The String Work Area

With functions using strings as parameters care is needed to look after the string work space that Blitz 2 uses to pass strings.

With the string function d$=Replace$(a$,b$,c$)

* Blitz 2 evaluates a$,b$ and c$ into the string workspace

* Pointers to a$,b$,c$ are in d0-d2

* lengths of the strings are at -4(a2),-8(a2) & -12(a2)

* The pointer to a$ will also be the pointer to the beginning of the string workspace.

* Register a2 will point to usable work space, if your function does not return a string i.e. len(a$) a2 should be returned pointing to the beginning of the workspace.

* d$ should be calculated using a3 as the start of the usable workspace and then copied to the beginning of the workspace

* set a3 to the end of d$ and make d0.l=length then return

MemLib

If you need to allocate memory in a routine use the functions available in MemLib. These are the same as Exec's AllocMem and FreeMem but Blitz 2 will remember the allocations and free up the memory you allocate at the end of the program or if the program is interrupted.

The ALibJsr routine is one of Blitz 2's Machine code to BASIC interfacing commands, generating a JSR command to the specified token number (or name).

To allocate memory use the following code in you routine...

```
move.l #numberofbytes,d0
move.l #memtype,d1
ALibJsr $c002              ;global allocmem
```

and to free memory....

```
move.l #addressofmemory,a1
move.l #numberofbytes,d0
ALibJsr $c003              ;global free mem
```

LibMacs Program Listing

The following is a complete listing of the libmacs file for reference purposes.

```
;
;pchk{a,b}
;
Macro pchk
CNIF `1<>`2
   CERR "Wrong number of macro parameters"
CEND
End Macro

;echk{a,b}
;
Macro echk
CNIF `1<`2
   CERR "Not enough parameters"
```

```
CEND
End Macro

;syslibheader{libnum,firsttoke,lasttoke+1,init,return,finit,error(,flags)}
;          1     2     3     4  5  6  7  8
Macro syslibheader
CNIF `0<>8
   !pchk{`0,7}
   MOVEQ #0,d0:RTS:Dc `1:Dc.l 0,0:Dc `2,`3:Dc.l 0,`4
   Dc `5:Dc.l `6,0,0,0,`7
CELSE
   MOVEQ #0,d0:RTS:Dc `1:Dc.l 0:Dc 0,`8,`2,`3:Dc.l 0,`4
   Dc `5:Dc.l `6,0,0,0,`7
CEND
End Macro

;libheader{libnum,init,return,finit,error}
;         1   2   3    4   5
Macro libheader
!pchk{`0,5}
MOVEQ #0,d0:RTS:Dc `1:Dcb.l 4,0:Dc.l `2:Dc `3:Dc.l `4
Dcb.l 3,0:Dc.l `5
End Macro

#wordret=-1:#noret=0:#longret=1

;libfin{}
;or libfin{toke,load,save,use,free,defmax,shift} if maximums needed for this
;         1   2 3   4   5   6   7
Macro libfin
CNIF `0>0
   !pchk{`0,7}
CEND
Dc -1
CNIF `0=0
   Dc.l 0
CELSE
   Dc.l `1,`2,`3,`4,`5:Dc 0,`6,`7
CEND
End Macro

;astatement
;
Macro astatement
!pchk{`0,0}
Dc 1,0,0
End Macro

;ustatement - unknown type, gets passed to sub on the stack
;
Macro ustatement
!pchk{`0,0}
Dc 8+1,0,0
End Macro

;afunction{type}
```

```
Macro afunction
!pchk{`0,1}
Dc {`1 LSL 8}+2,0,0
End Macro

;ufunction - unknown type, type selected pushed on stack
;
Macro ufunction
!pchk{`0,0}
Dc 8+2,0,0
End Macro

;acommand{type}
;
Macro acommand
!pchk{`0,1}
Dc {`1 LSL 8}+3,0,0
End Macro

#byte=1
#word=2
#long=3
#quick=4
#float=5
#string=7
;
#usesize=0
#unknown=8
#array=32
#varptr=128

;putargs{1,2,3,4}
;
Macro putargs
CSIF "`1">""
   Dc.b `1:!putargs{`2,`3,`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e}
CEND
End Macro

;args{1,2,3,4,...}
;
Macro args
Dc `0:!putargs{`1,`2,`3,`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e,`f}
Even
End Macro

;repargs{first rep,num rep,1,2,3,4.....}
;
Macro repargs
!echk{`0,3}
Dc {`1 LSL 12} OR {`2 LSL 8} OR {`0-2}
!putargs{`3,`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e,`f,`g,`h,`i,`j,`k}
Even
End Macro

;putlibs
```

```
Macro putlibs
CSIF "`1">""
  Dc `1,`2:!putlibs{`3,`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e,`f}
CEND
End Macro

;libs{lib,reg,lib,reg....}
;
Macro libs
!pchk{(`0 AND 1),0}
!putlibs{`1,`2,`3,`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e,`f,`g,`h}
Dc 0
End Macro

#intuition=255
#graphics=254
#exec=253
#dos=252
#diskfont=251
#graphics2=250
#exec2=249
#dos2=248

;These constants are for passing data directly to a register
;from a library.
;
#ld0=0:#ld1=$100:#ld2=$200:#ld3=$300
#ld4=$400:#ld5=$500:#ld6=$600:#ld7=$700
;
#la0=$1000:#la1=$1100:#la2=$1200:#la3=$1300
#la6=$1600

;This one means you want it pushed on the stack
;
#lpush=$ff00

;Asking for a USED type data puts the currently used
;struct of a max type lib into the appropriate reg
;
#used=2
;
#ud0=#ld0|#used:#ud1=#ld1|#used
#ud2=#ld2|#used:#ud3=#ld3|#used
#ud4=#ld4|#used:#ud5=#ld5|#used
#ud6=#ld6|#used:#ud7=#ld7|#used

#ua0=#la0|#used:#ua1=#la1|#used
#ua2=#la2|#used:#ua3=#la3|#used
#ua6=#la6|#used

;Asking for a BASE type data puts the base item (item 0)
;of a block of max type structs into the paritcular reg
;
#base=1
;
#bd0=#ld0|#base:#bd1=#ld1|#base
```

```
#bd2=#ld2 | #base:#bd3=#ld3 | #base
#bd4=#ld4 | #base:#bd5=#ld5 | #base
#bd6=#ld6 | #base:#bd7=#ld7 | #base
;
#ba0=#la0 | #base:#ba1=#la1 | #base
#ba2=#la2 | #base:#ba3=#la3 | #base
#ba6=#la6 | #base

;Asking for ITEM type data calculates a max type struct
;entry based on a specified data register. Just OR
;in the paramter (#pd0-#pd7) you which to use in the
;calculation. The final pointer ends in an address register
;
#pd0=0:#pd1=1:#pd2=2:#pd3=3
#pd4=4:#pd5=5:#pd6=6:#pd7=7
;
#item=$80
;
#ia0=#la0 | #item:#ia1=#la1 | #item
#ia2=#la2 | #item:#ia3=#la3 | #item
#ia6=#la6 | #item

;Asking for a MAX type data passes the currently defined
;maximum setting as selected in COMPILER OPTIONS
;
#max=3
;
#md0=#ld0 | #max:#md1=#ld1 | #max
#md2=#ld2 | #max:#md3=#ld3 | #max
#md4=#ld4 | #max:#md5=#ld5 | #max
#md6=#ld6 | #max:#md7=#ld7 | #max
;
#ma0=#la0 | #max:#ma1=#la1 | #max
#ma2=#la2 | #max:#ma3=#la3 | #max
#ma6=#la6 | #max

;subs{code,error1,0}
;
Macro subs
!pchk{`0,3}
Dc.l `2,`1,`3
Even
End Macro

;name{"name","help"}
;or name{"name","help",label}
;
Macro name
CNIF `0<>2
   !pchk{`0,3}
CEND
Dc -1:Dc.l 0:Dc 0
`3:Dc.b `1,0
Dc.b `2,0
Even
End Macro
```

```
;nullsub{routine,error1,error2,lib,lib,lib....}
;
Macro nullsub
!echk{`0,3}
Dc 0,0,0
!libs{`4,`5,`6,`7,`8,`9,`a,`b,`c,`d,`e,`f}
Dc.l `2,`1,`3
End Macro

;dumtoke{"name","help",label}
;
Macro dumtoke
!pchk{3,`0}
Dc 8,0,0:Dc.l 0:Dc 0
`3:Dc.b `1,0
Dc.b `2,0
Even
End Macro

Macro blitwait
BTST #6,$002(a1):'blitwait`@:BTST #6,$002(a1):BNE 'blitwait`@
End Macro

;copwait{x,y}
;
Macro copwait
!pchk{2,`0}
Dc {{`2 LSL 8} OR `1}+7,$fffe
End Macro

;copmove{reg,val}
;.
Macro copmove
!pchk{2,`0}
Dc `1,`2
End Macro

;copend
;
Macro copend
!pchk{0,`0}
Dc $ffff,$fffe
End Macro

;
;OK, now for all then Blitz library numbers
;
;first, system ones...

#arrayslib=65434
#chipbaselib=64100
#clrlib=64635
#datalib=64535
#errtraplib=64090
#exitslib=65035
#ffplib=64935
```

```
#floatquicklib=64080
#intlib=65520
#ldivlib=64735
#lmullib=64835
#maxlenlib=64135
#maxslib=64335
#memlib=65530
#modlib=64070
#runerrlib=65500
#staticslib=65335
#strcomplib=64435
#switchlib=65510
#varslib=65534
#wbstartuplib=65533
#strings1lib=65235
#strings2lib=65135
;
;Amiga library ones...
;
#intuitionlib=255
#graphicslib=254
#execlib=253
#doslib=252
#diskfontlib=251
#graphics2lib=250
#exec2lib=249
#dos2lib=248
;
;now, basic ones...
;
#o=55 ;offset
;
#sis2dlib=174-#o
#audiolib=171-#o
#bblitlib=201-#o
#bitmaplib=215-#o
#blitlib=205-#o
#blitzcoplib=199-#o
#blitzkeyslib=153-#o
#brexxlib=187-#o
#cliargslib=161-#o
#collslib=159-#o
#editlib=241-#o
#fadelib=157-#o
#fileiolib=179-#o
#fontlib=197-#o
#freqlib=207-#o
#gadgetslib=195-#o
#gameiolib=245-#o
#iffiolib=217-#o
#iffmakelib=169-#o
#ilbmifflib=216-#o
#inputoutputlib=219-#o
#intuifontlib=189-#o
#linklistlib=183-#o
#mathtranslib=247-#o
```

```
#menuslib=191-#o
#mouselib=163-#o
#palettelib=211-#o
#printlib=242-#o
#qblitlib=203-#o
#rawkeylib=185-#o
#screenslib=213-#o
#scrolllib=155-#o
#shapeslib=209-#o
#shapetrixlib=173-#o
#sortlib=165-#o
#spriteslib=167-#o
#stencillib=177-#o
#stringfunclib=240-#o
#trackerlib=151-#o
#vallib=175-#o
#windowslib=193-#o
#memacclib=235-#o

;Setting up a library should happen as follows:
;
;any dumtokes for object names should be at the top
;(after libheader), so they don't get moved around when
;the library is added too.
;
;  !libheader
;
;  (!dumtoke) for object name
;
;  !astatement/!function/!ustatement/!ufunction
;  !args
;  !libs
;  !subs (back to args/libs if more)
;  !name
;
;  !astatement
;  .
;  .
;  .
;
;
;  !dumtoke
;  .
;  .
;  .
;
;
;  !nullsub
;  .
;  .
;  .
;
;
;  !libfin

;
```

```
;
;Then your actual code!
```

Conclusion

We have not covered half of the features of the Blitz library system in this chapter, to do a good job we would have had to devote a whole book to the subject. Becoming a registered user of Blitz will ensure you receive more specific documentation regarding the more indepth features when developing libraries.

6. About the Amiga

The following is a brief description of both the Amiga operating system and its powerful hardware. These two subjects are both quite advanced but suprisingly very much independent.

Applications and productivity software developers are interested in interfacing to the Amiga operating system in a clean and well behaved fashion while game developers have the goal of squeezing every drop of speed out of the Amiga's hardware viewing the operating system as one large obstacle in their quest for flicker free action packed code.

A lot of effort has gone into making Blitz 2 the ideal environment for both types of development. There is some mutual ground to be covered, games programmers will find linked lists more efficient than standard arrays in both speed and versatility and applications programmers now value speed as a serious bottleneck in developing true 'productivity' software, especially now graphics environments are here to stay.

The following should be treated as background reading, the concepts are important to a greater understaing of the Amiga but their implementation is all but taken care of by the Blitz 2 environment.

The Amiga Operating System

The Amiga Operating System is a very advanced multitasking environment. This means that although the Amiga has only one main micro-processor the machine can seemingly run many tasks simultaneously.

EXEC

The core of the Amiga OS is a multitasking kernal named EXEC. When the Amiga is first turned on, EXEC fires up, registers all the free memory in the system, initialises all the devices connected to the Amiga such as the keyboard, floppy disk etc and then sets about launching the operating system includeing the libraries such as DOS and Intuition.

Unlike many computers, the software developer on the Amiga needs to learn that they are not king of the jungle (unless they enter Blitz mode that is!). EXEC is in charge, and it is EXEC who makes the rules.

EXEC keeps control of the system using large networks of linked lists. This system means EVERYTHING is accounted for and interconnected. It delivers a high level of flexibility enabling complex systems to live in the same environment sharing the same processor, memory and resources with ease.

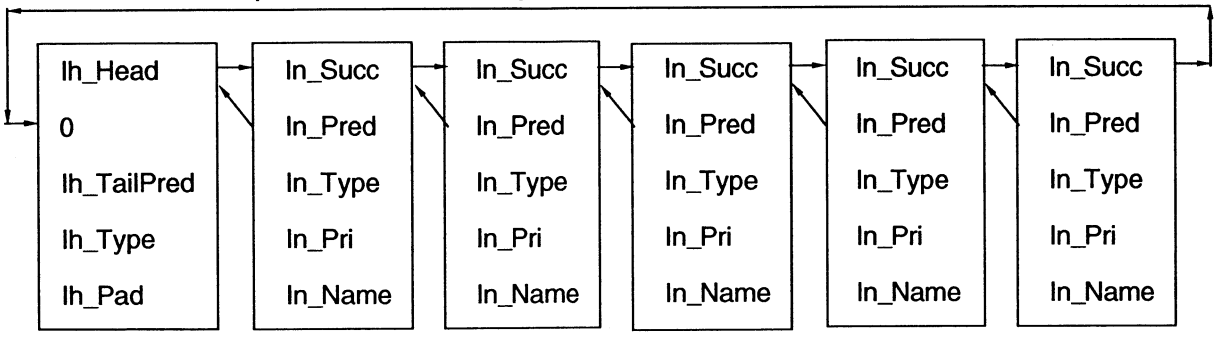
EXEC: Linked Lists

A list is a collection of items containing information about the same topic. By keeping this list in a sequential order in memory (such as an array) the computer can keep control of the data. However the task of removing and adding items to a list held sequentially is very difficult and time consuming. By

dynamically allocting memory for each item and connecting it with links to the other members of the list, EXEC's linked lists are a clean, efficient way of maintaining unknown quantities of information.

Each item of information has a header called a node. This node has two 'pointers' one to the item that comes before it in the list (pred) and one that points to the next item in the list (succ).

A List header is a special node without any information connected to it.



As an example the following is a short program that prints out all the devices connected to the system.

The AmigaLibs.Res file should be resident by adding its name to the resident list in the Options requester.

```
;
; Exec List processor
;
*exec.ExecBase=Peek.l(4)

*mylist.List=*exec\DeviceList   ;also try LibList and any other Lists in ExecBase

*mynode.Node=*mylist\lh_Head

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

MouseWait
```

To delete an item from a list simply make the previous node's ln_succ point to the next node, and make the next nodes ln_pred point to the last node. Your node is now no longer part of the list and its memory can be freed up.

To add an item, simply allocate its memory, for sorted lists find the position in the list you wish to add the item or for unsorted locate the lh_head node and rearrange the ln_succ and ln_pred pointers accordingly of both your node and the node you are inserting in front of.

Note: because lists are often shared by other programs or interrupt handlers you may have to issue a Forbid before tampering with a list's contents.

EXEC: Signals

Multitasking means EXEC needs to keep control of many programs at once, allocating them processor time, memory etc. tasks made simpler due to the friendly nature of its linked lists.

Signals allow a program to tell Exec that it is not actually doing anything, and does not have to be multi-tasked (given a share of the processor's time) until some event takes place. Events such as the user selecting a menu or gadget.

It must be noted that the ability for a program to wait for events is really down to the programmer. He/She must make use of the appropriate commands in whatever language they are using to achieve this efficient multi-tasking. In AmigaBasic, this command was 'Sleep'.

In Blitz 2, it is 'WaitEvent'. Failure to use these commands will mean a program will always be multi-tasking (using processor time), even it is doing nothing but waiting for a keystroke.

The following are two implementations of a routine that waits for a keystroke. The first will slow all other programs down while it continuously calls the inkey$ command while the second goes to sleep until a signal from EXEC wakes it up.

example 1:

```
Repeat i$=Inkey$ Until i$>""
```

example 2:

```
Repeat WaitEvent:i$=Inkey$ Until i$>""
```

The WaitEvent command in the second example is a means of telling Exec not to bother about multi-tasking our program, until an event takes place. Our program will therefore not be slowing down other programs while it is waiting for it's keystroke. Please refer to the WaitEvent command in the Windows chapter of the reference section for more information on WaitEvent.

EXEC: Libraries & Devices

The Amiga's operating system is a combination of EXEC a small kernal and a whole host of routines available to the programmer in the form of Libraries & Devices.

To use DOS the programmer must first open the DOS library via an EXEC call, EXEC will then return the location of the DOS library in memory and the program is now able to call DOS routines such as Open and Write by jumping to offsets from the DOS library location with the appropriate 68000 data registers loaded with parameters.

Blitz 2 does the above work for you, and all the routines included in these libraries have been 'tokenised' so they are available as BASIC keywords.

*see the details of LibConvert for including third party Amiga Libraries into the Blitz 2 environment.

Devices are similar to libraries in that it is up to EXEC to install them in memory and return pointers to your software for you to access them. However they have a standard set of commands and information is passed to them in packets (known as an IO Request) rather than 68000 data registers. There are examples on the disks that come with Blitz 2 showing correct use of EXEC devices.

EXEC: Other Information.

Because of the way Blitz 2 handles Messages, Ports, Interrupts and Memory Allocation these topics will not be discussed here. Suffice to say that Blitz 2 simplifies the use of these tools dramatically and EXEC's complex array of linked lists is in the background dealing with these problems with seemingly elegant ease.

Intuition

The most visible thing about the Amiga has to be its graphics user interface known as Intuition. Intuition's main task as it's name states is to make the operating of the Amiga as intuitive as possible.

A program's user-interface refers to how a program represents itself to anyone wishing to use the program. This includes such things as how the various options of a program are presented, how data is entered into a program, and how output data is presented by a program.

As a software developer on the Amiga you are encouraged to use Intuition to its maximum, this ensures a continuity between different software packages making the learning curve of the average Amiga user as gentle as possible.

Following is a description of the main ingredients that make up Intuition.

Screens

A Screen is one of Intuition's basic building blocks. An example of an Intuition screen is the Workbench screen. Any program which uses Intuition must reference a screen of some kind, whether it be the Workbench screen, or the programs own 'custom' screen. A screen itself is not of much use. Normally, at least one Intuition window will have to be opened on the screen to allow your program Intuition style inputor output. Screens have the following features and limitations:

* A screen is identifiable by it's title bar and depth arrangement gadgets. A screen's title bar is also where any menu options appear.If there are no windows open on a screen, the screen will appear to be 'blank' below the title bar.

* Screens may be dragged vertically by grabbing the title bar with the mouse pointer. Under the 2.0 opertaing system, screens may also be dragged horizontally to a certain degree.

* Screens may be organized to appear in vertical order, but never in horizontal order. This means that although screens may be positioned 'on top of' each other, than can never be positioned 'beside' each other. This limitation is due to the Amiga's 'copper' hardware which is used to display screens.

* Each screen may appear in it's own display mode, with it's own palette and with it's own number of colours.

Windows

Windows are probably the most important of Intuition's elements. An example of a window is a CLI window. A window can be thought of as an independant 'input' terminal, through which the Amiga's user may enter data, select menus or click on gadgets. Here are some of the features and limitations of windows:

* A Window must appear on a screen, and are always rectangular in appearance. Windows may contain various system gadgets including a drag bar and sizing gadget, and may have title text. A drag bar allows a window to be moved around within it's screen, and a sizing gadget allows the size of the window to be altered. A window may never be positioned or sized outside of it's screen.

* Windows may also contain a 'depth arrangement' and/or 'close'gadget, allowing the user to arrange Amiga windows in order of preference, and to close down windows.

* Windows may be either 'active' or 'inactive'. Inactive windows are identified by 'ghosted' drag-bar imagery. Only one window may ever be active at a time. To make a window active, simply click anywhere inside the window with the left mouse button. This will 'de-activate' the previously active window, and cause the specified window to become active.

* Windows may have custom menus and/or gadgets attached them. When the Amiga's right mouse button is pressed, the menu associated with the current active window is display across the title bar of the window's screen.

* Windows may also be opened without any gadgets, borders or titletext at all. A window opened like this will appear not to have opened at all! This allows you to attach menus and gadgets to an apparently 'empty' screen. This is in fact what the WorkBench program does.

Menus

Menus are a very useful way of presenting a list of options to a program user. An example of a menu is the WorkBench menu available when WorkBench is loaded. You access menus by holding down the rightmouse button. If a menu is available, a main list of options will appear horizontally across the top of the screen. By still holding down the right mouse button and positioning the mouse over one of these options, further options will appear vertically below the selected main option. To choose one of these options, position the mouse over the desired option, and release the right mouse button. Menus have the following features:

* As menus are attached to windows, the currently active window will determine which menu is presented when the right mouse button is pressed.

* The options that appear horizontally across the top of the screen are known as menu titles. The options which appear vertically below these are known as menu items. Menu items may even have further subitems, which are normally presented vertically below and to the right of the menu item they are attached to.

Gadgets

A gadget may be likened to a push-button. An example of gadgets are the icons WorkBench uses to represent various disks available. Gadgets are normally selected using the left mouse button. There are two main types of gadgets - system gadgets and custom gadgets. System gadgets are gadgets supplied by Intuition - for example, the screen depth arrangement gadgets, or the window sizing gadget. Custom gadgets are gadgets supplied by a programmer. There are three main types of custom gadget available. Boolean gadgets, string gadgets and proportional gadgets:

* Boolean gadgets are simple one-click-to-activate gadgets, and are most commonly used for the 'OK' or 'CANCEL' gadgets found in many programs. Boolean gadgets may be represented by the use of text or of graphical imagery.

* String gadgets allow a line of text to be entered, and can be used to receive simple user input. These gadgets are often seen on file requesters, in the area where the path and file name are entered.

* Proportional gadgets are a little more complex. These gadgets are in the form of a 'slider bar' which allows the user to alter a 'setting' of some kind. Proportional gadgets are often found in art packages to create the RGB sliders found in colour selectors, each of the sliders altering the setting of one of the RGB components. Proportional gadgets can be in the form of either vertical or horizontal sliders, or even a combination of both.

Intuition Conclusion

Using Intuition can be a complex task from C or assembler. However from Blitz 2 windows and screens can be opened with one call. Menus and gadgets can be linked to windows simply and easily and user input can be monitored with the minimum of fuss.

It should be stressed that when designing your user interface Commodore has set some constructive

guidelines concerning menu lay out etc. The main points are as follows

- use OffMenu when an item becomes non-functional

- to show a submenu exists of a main menu use chr$($BB) ">>"

- if applicable always put the PROJECT menu at the left in the order: NEW OPEN SAVE SAVE AS PRINT PRINT AS ABOUT QUIT

- if you have an EDIT menu put it next with the following order: UNDO CUT COPY PASTE ERASE

- with requeters put OK on the left and CANCEL to the right

- use left mouse button for selection right button for information transfer

- icons should be designed with lightsource coming from top left

- keep it simple but elegant

The Hardware

What really lies behind the power of the Amiga is, of course, it's hardware. The Amiga's hardware includes all the 'nuts and bolts' parts inside the machine, and is responsible for such low level things as graphics and sound.

There are also the more esoteric hardware elements, such as the infamous 'blitter' chip and the simple graphics coprocessor. The hardware represents the practical limits of what a program can achieve. For example, it is pointless writing a program to handle a 160 character wide display if the hardware is only capable of producing character displays up to 80 characters wide.

A basic understanding of the Amiga's hardware will help you understand what the Amiga is capable of, and will also help in understanding the logic of many Blitz 2 commands.

Pixels

Everything that appears on the Amiga's display is actually made up of thousands of little dots laid out in a grid-like manner. An individual dot is known as a 'pixel'. If all pixels on the display a reset to the same colour, the display appears to be blank.

When a group of pixels is set to a different colour, this group will appear to take on a shape. This is how all shapes, be they characters, space invaders or mouse pointers, are created on the Amiga. To alter the display in any way, you often have to specify which pixel position you wish to alter.

Blitz 2 uses a co-ordinate system to do this, where two numbers are used to specify a pixel position. One number specifies an'across the pixel grid' ('x') position, while the other number specifies a 'down the pixel grid' ('y') position. Pixels further to the right across the display have higher 'x' coordinates, and pixel further down the display have higher 'y' coordinates.

Commands in Blitz 2 which require a pixel position always expect the 'x' coordinate first, then the 'y' coordinate.

RGB Colour

The Amiga's colour capabilities are one of it's many impressive features. The Amiga is capable of displaying 4096 different colours, but how do you go about selecting one particular colour? The Amiga

uses a Red/Green/Blue (RGB) system to do this. Any of the Amiga's 4096 colours can be described by 'mixing' quantities of these three colours (known as 'primary' colours) together.

For example, to get the colour purple, you mix red and blue together. To get the colour yellow, you mix red and green together. To get the colour orange, you also mix red and green together, but use more red than green.

The Amiga lets you select from sixteen different quantities (brightnesses) of each of the primary colours to arrive at a desired colour. Zero repesents the lowest (darkest) quantity, and fifteen represents the highest (brightest) quantity. Therefore, to select the colour purple, we could select a red brightness of fifteen, a green brightness of zero, and a blue brightness of fifteen. Here are some example mixes of primary colours:

RED	GREEN	BLUE	RESULTING COLOUR
15	15	15	White
8	8	8	Grey
15	15	0	Yellow
15	8	0	Orange
4	0	0	Dark Red
0	0	0	Black

Since there are sixteen possibles brightnesses of the three primary colours, there are 16 to the power of three, or 4096 different colour combinations available.

Colour Palettes

Although there are 4096 colours available on the Amiga, there is normally a limit on how many of these colours may be displayed on the screen at once due to the Amiga's 'palette' system.

The Amiga's palette consists of up to 32 'colour registers', each of which may be set to any RGB colour. When it comes to selecting what colour a pixel on the display should be, you don't specify an actual RGB colour, instead you specify one of the palette's colour registers. Therefore, you can normally only have up to 32 different colours on screen at once (I say 'normally' because there are various tricks which can to some extent get around this limitation).

Let's say you wanted to set a pixel on the display to green. To do this, you would need to first set one of the colour registers to green, then specify that colour register when plotting the pixel.

Bitplanes

Bitplanes are another fundamental part of the display, as they "hold the pixels" that make up everything on the screen. If we breakdown the word BitPlanes into it's two parts, we can get a rough idea what they are about.

The first part, "Bit", is a unit of a computers memory as well as other things, and there are 8 bits to a byte. These bits when overlapped, correspond to one pixel on the screen.

The second word "Planes" is an idea of how to visualise the concept of the display. The screen is made up of "overlapping" planes of memory, that when combined, produce the correct pixel colours on

the screen. So BitPlanes can be though of as a number of overlapping groups of bits that make up the display. Now to get a certain number of colours on the screen, you need to have multiple bitplanes. Each time another bitplane is added to the display, we are adding another "bit" and therefore doubling the number of colours that we can use.

A 3 bitplane screen will give us 8 colours (combinations)

The same bit from each bitplane is combined to create the bits needed for the pixel. So if you had 3 bitplanes and set all of the first bits of the first byte of each bitplane, then you would be generating colour number 7.

Display Modes

The Amiga is capable of displaying graphics in a variety of ways. These 'display modes' affect such things as the number of colours available in the display palette, the size of the pixels on the display, and how colours are used on the display. In this section we shall look at each of the Amiga's display modes in turn:

High Resolution (hi res)

Hi res mode is the display mode the Amiga boots up in. Hi res mode gives you a pixel resolution of about 640 pixels across by 200 pixels down. In hi res mode, displays may be from one to four bitplanes deep.

Low Resolution (lo res)

Lo res mode gives a 'chunkier' display. Each lo res pixel is twice as wide as a hi res pixel. This gives about 320 pixels across by 200 pixels down. In lo res mode, displays may be from one to five bitplanes deep. Another bitplane can be added, but then the display becomes another special mode.

Interlaced

Interlaced mode doubles the height of the screen. This gives you 400 lines and either 640 or 320 pixels across, depending on what other resolution you use with interlaced. Likewise the colours avaiable are determined by the other mode used.

Extra Half-Brite

This mode can be used only with lo-res or lo- res/interlaced screens. The number of bitplanes used must be 6. This extra bitplane would normally give you a maximum of 64 colours to use. It does, but it also has a restriction.

The top 32 colours that you can use are not definable, as there are only 32 colour registers available after all. The top colours will always be "Half as Bright" as the top colour-32. Eg colour 32 will be half as bright as colour 0, colour 54 will be half as bright as colour 22 etc.

Hold and Modify (HAM)

This is another special amiga display mode that allows 4096 colours to be displayed all at once on the screen. Like Extra Half Brite, it uses 6 bitplanes also, but the 6th bitplane is used differently. Here is how the bitplanes are interpreted for the HAM mode.

The upper 2 bits (bits 4 and 5 (from bitplanes 5 and 6)) determine how the lower four bits are decoded. If the upper 2 bits are both 0, then the colour is taken directly from colour registers 0 to 15 (16 colours) so you can select 16 colours directly using this method.

When the 2 upper bits are not 0, then the colour value of the pixel to the left is taken and one component (Red, Gree or Blue) of that colour is changed and placed in the pixel position. The value

for this new compnent is got from the lower four bits. The 2 upper bits determine what component is changed. Here is a table showing the upper bit combinations and what component they change.

5	4	3	2	1	0	Affect in Hold and Modify Mode
0	0	P3	P2	P1	P0	Colour taken from palette
0	1	B3	B2	B1	B0	Left Pixel's Colour with Blue Modified
1	0	R3	R2	R1	R0	Left Pixel's Colour with Red Modified
1	1	G3	G2	G1	G0	Left Pixel's Colour with Green Modified

Dual Playfield

Dual playfield is another of the amigas display mode, but it has been put in it's own section as it is quite different from the other modes like lo-res, HAM etc.

Dual playfield allows the overlapping of two completely independent playfields or screens. These playfields can be scrolled independently, and can be of different sizes. Unfortunately there is a disadvantage with this mode.

As the Amiga can not display any more than 6 bitplanes, the 6 bitplanes available have to be divided amongst the two playfields. Each playfield therefore gets 3 bitplanes, allowing for a maximum of 8 colours per playfield. The division of these bitplanes is not simply bitplanes 1,2 and 3 are for playfield 1, and 3,4,5 for playfield 2. The division is odd bitplanes for playfield 1 and even bitplanes for playfield 2. Colours for the playfields come from the colour registers as usual, except colour registers 0-7 are for playfield 1 and registers 8-15 are for playfield 2.

Now you don't have to use all 6 bitplanes, you can use only 2 if you wish, but because of the way they a distributed interleaved odd/even some restrictions apply. Here it a table showing the number of bitplanes you may want to use and their distribution.

No. of BitPlanes	PF1 BitPlanes	PF2 BitPlanes
1	1	none
2	1	2
3	1,3	2
4	1,3	2,4
5	1,3,5	2,4
6	1,3,5	2,4,6

The Blitter

The Blitter is used to tranfer blocks of memory from one place to another. It can actually read from 3 separate areas of memory (known as channels A,B and C) and depending on the combination of each will output a result to channel D.

The MinTerm is used to specify what logic the Blitter should perform on the 3 Source blocks of memory to generate the resulting Destination block.

The Blit commands in Blitz 2 uses channel A for what is known as a cookie cut, B for the shape data, and Cfor the destination memory. Thinking of the cookie data as a stencil we want the Blitter to transfer our shape data where the stencil is on and use the destination data when it is blank.

The logic function we use can be put logically as Use B channel when A channel is on otherwise use the C channel. The Hardware appendix in the Blitz 2 reference manual discusses this topic in more detail.

The Blitter is also able to 'barrel shif't the data in A and B as well as Mask the first and last words of the A channel. Again more information detailing the Blitter chip can be found in the hardware appendix of the reference manual.

The Copper

Copper stands for Co-Processor, which is exactly what the Copper is. It is another processor that will work in conjunction with the main processor

It's main task is to sustain the actual display. It sets up the colour palette, and screen width, height, mode etc. It can do a few other thing as you will realise later when we look at it's instruction set, but mainly it is for the display handling.

To program the copper, a series of words is placed in memory and the copper pointer registers are set to point to this area. Then the copper is started with a move to a COPJMP register.

The Copper doesn't have the flexible command set like the 68000, it is endowed with only 3 instructions, but these instructions do the job that they are required to.

* MOVE

This instruction will move a value into a hardware register, it would be the most widely used instruction that the copper has.

* WAIT

This instruction will halt the Copper's execution until the raster beam reaches or is past the y position specified in the instruction.

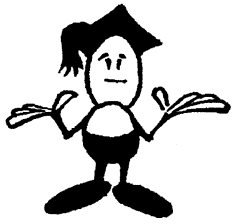
* SKIP

Skip checks the vertical beam position, and if it is greater than or equal to than that specified, the next instruction is skipped, other wise it is executed. This instruction could cause the copper tostart executing another different copper list. It is like aconditional branch.

Summary

The detail in this last chapter was intended as a brief overview of the Amiga, it's Operating system and its Hardware. There are many excellent reference books relating to the topics we have quickly skimmed over here that cover the topics in much greater detail.

7. Tutorials



The following are a selection of small programs that will introduce the core commands and techniques needed to program in Blitz 2.

1. Input/Output with Blitz 2.

The following three programs detail the commands needed to print and input information using Blitz 2.

The first example uses the default console window. When a program is 'RUN' from the Blitz 2 editor a console window is opened up. If this program is made an executable and run from the CLI it will use the CLI window.

The first 3 lines are called remarks. Blitz 2 uses the semicolon instead of the REM statement for remarks. When Blitz 2 finds a semicolon on a line of code it ignores the rest of the line, this enables the programmer to put useful comments and headings into the code which will then be ignored by the compiler.

The **NPrint** statement is the same as **Print** but generates a linefeed at the end of the line, effectively placing the cursor on the next line, so the next **Print** or **NPrint** statement starts on a new line.

Blitz 2 uses the **Edit$()** function instead of the standard BASIC Input statement. This is because Input is a function and the Blitz 2 compiler requires that all functions return a value that is assigned to the variable before the = sign.

Edit$() requires a parameter that states the maximum number of characters allowed to be read into the string. An optional parameter can specify a default string which will be printed and if the user just hits [Return] will be returned as the user's answer.

The **MouseWait** command will wait until the user presses the left mouse button before continuing.

```
;
; default window IO example
;
NPrint "Welcome!"                ;print and move cursor to next line
Print "What is your name? "       ;leave cursor at end of line
a$=Edit$(40)                     ;input name max 40 chars
Print "Press left mouse button please ",a$  ;print message and name
MouseWait                        ;wait for mousebutton down
End
```

The next program opens it's own Intuition Window which it uses to input and output information.

The **FindScreen** command will locate the front most Screen being displayed by Intuition and assigns it to the Blitz 2 Screen Object number specified. An optional parameter can specify the title of the Screen to be used.

The **Window** command opens a Window on the specified Screen. For a full description of its parameters please refer to the Blitz 2 Reference Manual.

Note that we do not have to close the Window as Blitz 2 will automatically close it for us when the

Program ends.

```
;
; custom window IO example
;
FindScreen 0      ;get front most screen
Window 0,0,10,320,190,$143f,"Hello World",1,2  ;open a window
WLocate 0,0       ;position cursor in window
NPrint "Welcome!"  ;print and move cursor to next line
Print "What is your name? "    ;leaves cursor at end of line
a$=Edit$(40)              ;input from user
Print "Press left mouse button please ",a$  ;print message and name
MouseWait                         ;wait for mouse button
End
```

The next program is an example of Input and Output in Blitz mode. Because Blitz mode is designed for maximum performance there is a little more code required to 'configure' Blitz mode to set up a display, accept keyboard events and print information.

BLITZ tells the compiler we are now in Blitz mode. It will turn off the operating system and shut down all interrupts.

BlitzKeys On sets up Blitz's own keyboard handler which will keep track of keys pressed by the user.

The **BitMap** command allocates enough memory for a display 320x256 pixels, with 2 colours. This is a standard lo-res pal display.

BitMapOutput 0 tells Blitz to use BitMap 0 for all following print statements.

BitMapInput tells Blitz that keyboard events will be required and should be echoed on the current BitMap when using the edit$() function.

The **Slice** command sets up a copper list to drive the Amiga display. Slices configure the Amiga's video output so that each frame, the Amiga knows what resolution and how many colours it needs to display as the video beam draws line after line down the monitor.

The **Show** command tells the current Slice which Bitmap to use to get pixel information from. The RGB command sets palette information for the currently used slice.

```
;
; Blitz mode IO example
;
BLITZ          ;Blitz mode ON
BlitzKeys On      ;turn Blitz keybord handler on
BitMap 0,320,256,1 ;define monochrome bitmap
BitMapOutput 0    ;output prints to bitmap 0
BitMapInput       ;use same bitmap for edit$
Slice 0,44,1            ;a slice for our display
Show 0                  ;show bitmap 0 on our slice
RGB 1,15,15,15          ;set colour 1 to white
NPrint "Welcome!"         ;print and move cursor to new line
Print "What is your name? "  ;leave cursor at end of line
a$=Edit$(40)              ;get user input
NPrint ""                 ;print a new line
Print "Press left mouse button please ",a$  ;print message and name
MouseWait
End
```

2. Simple Graphics with Blitz 2

The next two programs illustrate using the simple graphics commands in both Blitz and Amiga modes.

The constants #w and #h are defined so the screen width and height parameters used throughout the program can be easily changed.

For speed, the variable i is defined as the word type. This is the fastest type and gives the variable i an integer range of -32768 to 32767.

The **CLS** command clears the screen.

The **Rnd()** function returns a number between 0 and the parameter supplied.

The window based commands **WPlot**, **Wline**, **WBox** and **WEllipse** clip their rendering so that only the area inside the window is drawn on, this explains why they are slower than their bitmap based counterparts.

```
;
; simple graphics in Amiga mode
;
#w=640:#h=180   ;setup constants for display size
DEFTYPE .w i    ;default type is word for speed
FindScreen 0    ;get front most screen and open a window
Window 0,0,10,#w,#h,$143f,"1000 Plots, Lines, Boxes and 20 Ellipses",1,2

WCls
For i=1 To 1000
 WPlot Rnd(#w),Rnd(#h),Rnd(7)
Next

WCls
For i=1 To 1000
 Wline Rnd(#w),Rnd(#h),Rnd(#w),Rnd(#h),Rnd(7)
Next

WCls
For i=1 To 1000
 WBox Rnd(#w),Rnd(#h),Rnd(#w),Rnd(#h),Rnd(7)
Next

WCls
For i=1 To 20
 WEllipse Rnd(#w),Rnd(#h),Rnd(#w),Rnd(#h),Rnd(7)
Next

MouseWait
End
```

The next example is similar but operates in Blitz mode and uses the bitmap based forms of the commands box, line, plot and circle.

The display used is hires 8 colour so the flags value in the slice command is 11: 8(hires) | 3(bitplanes).

```
;
; simple graphics demo in Blitz mode
;
#w=640:#h=180      ;define constants for width and height of display
DEFTYPE .w i       ;set default type to word for speed
BLITZ              ;into BLITZ mode
BitMap 0,640,256,3 ;setup bitmap
BitMapOutput 0     ;prints will goto bitmap
Slice 0,44,11      ;setup slice for display
Show 0             ;and show bitmap 0
Print "1000 Plots, Lines, Boxes and 2000 Ellipses!"

Cls
For i=1 To 1000
 Plot Rnd(#w),Rnd(#h),Rnd(7)
Next

Cls
For i=1 To 1000
 Line Rnd(#w),Rnd(#h),Rnd(#w),Rnd(#h),Rnd(7)
Next

Cls
For i=1 To 1000
 Boxf Rnd(#w),Rnd(#h),Rnd(#w),Rnd(#h),Rnd(7)
Next

Cls
For i=1 To 20
 Circle Rnd(#w),Rnd(#h),Rnd(#w),Rnd(#h),Rnd(7)
Next

MouseWait
End
```

Note:

Although the **For..Next** structure was used in the previous examples for simplicity's sake, a faster method to execute the loop would have been to use the **While..Wend** structure. To plot 1000 points with **While..Wend**, the code would look like this:

```
i.w=0
While i<1000
  Plot ....
  i+1
Wend
```

The speed increase is due to the **For..Next** structure needing to use the stack to store the limits and step variables where as a **While..Wend** loop does not. Such overhead is only noticeable when the inside of the loop contains little logic.

3. Intuition Screens, Menus and the Blitz 2 FileRequester.

The following program opens its own Intuition screen. It then configures a menu list containg menus for Load, Save and Quit with their appropriate keyboard shortcuts.

The **maxlen** statements are needed to set aside enough memory for the Blitz 2 file requester internal string space.

After opening an Intuition Window on our Screen the **SetMenu** command attaches our menu list to the window.

The **Repeat...Forever** structure is our main program loop.

The **WaitEvent** command puts our program to sleep so it does not hog processor time in the Amiga's multitasking environment. When the user selects a Menu, WaitEvent will wake up our program and continue execution so that we can take whatever steps are necesary to respond to the user's input.

By using the **Select..Case..Case..EndSelect** structure we can handle each situation in a tidy structured manner. **WaitEvent** will return the value 256 if indeed the event was a menu selection by the user.

The **ItemHit** function returns which menu item the user selected.

The **FileRequest$** function lets the user enter a filename which it returns. If the user cancels the file requester a null string is returned.

```
;
; 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$=192  ;These lines MUST be executed before a file requester is used
MaxLen name$=192

Window 0,0,0,320,200,$1900,"",1,2  ;This sets up a BACKDROP (ie - invisible) window
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$=FileRequest$("File to Load",path$,name$)
    NPrint "Attempted to Load ",p$

   Case 1  ;save    ;its item 1 which means save
    p$=FileRequest$("File to Save",path$,name$)
```

```
    NPrint "Attempted to Save ",p$

    Case 2          ;its item 2 which means quit
     End

   End Select

  End Select

 Forever
```

4. Prime Number Generator example.

The following program generates a list of prime numbers from 2 upto 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.

```
;
; find prime numbers example
;
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 out 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 it 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
End
```

5. Shapes and Blitting in Blitz mode.

The following program demonstrates how to smoothly animate shapes in Blitz mode. It sets up a List of shapes and utilises a double buffered display. Using queues the shapes are quickly erased from the current bitmap then the program goes through the list, moving and redrawing the shapes. By using two bitplanes, one is displayed while the other is redrawn resulting in a flicker free display of moving animated graphics.

Because each shape needs to have more than one piece of information stored about it we set up a **NewType** that has 4 'fields' to hold the shapes angle and distance (polar coordinates for this example), it's speed and animation frame.

Once we have defined the NewType planet which will hold these different values we dimension an array of type .Planet. This is a specail array known as a List. Being a List we can easily loop through all the planets currently held in the array as well as take advantage of the speed increase that Lists offer over standard arrays.

The **ResetList..While AddItem...Wend** structure is used to fill the List with different Planets. Because of the **USEPATH** compiler directive we have used, any reference to a current item's field is simply referred to with the \ symbol which is replaced internally with p()\ by the compiler.

The line \angle=Rnd,Rnd.. assigns the random parameters generated to the current item's angle, dist, speed and frame fields in one go. This is known as a multiple assign.

The subroutine makeshapes draws a few circles and lines to the current bitmap. It then uses **GetAShape** to turn the rectangle on the bitmap into a shape for blitting. Each shape needs a cookie for drawing as well as a central 'handle' as it will be rotating. The handle is an offset from the top, left of a shape that Blitz uses to position the shape when blitting.

By making 16 copies of our shape and rotating each one a little more than the last we end up with a series of 16 smoothly rotating frames.

The main loop first flips the display so the bitmap just drawn is now displayed and the other bitmap can be used for drawing. The **VWait** command is used to sync this double buffering with the video display.

The **UnQueue** command will remove the shapes blitted to the current bitmap (note that each bitmap needs its own queue). The **While NextItem..Wend** loop is used to step through each of the items in

the planets list. The **QBlit** command not only draws the shape to the screen but also adds the rectangular area affected to the Queue structure for the Unqueue that will happen in two frames time.

```
;
; shape handling in blitz mode example
;
num=5

BLITZ                ;go into blitz mode
BitMap 0,320,256,3   ;set up 2 bitmaps as double buffering
BitMap 1,320,256,3   ;shows 1 and updates the other
Slice 0,44,3         ;slice for a display

NEWTYPE .planet      ;need to hold angle distance speed
 angle.f             ;and animation information for each
 dist.f              ;planet
 speed.f
 frame.q
End NEWTYPE

Dim List p.planet(num-1) ;dim a list of planets

USEPATH p()          ;save typeing p()\.. all the time

Queue 0,num          ;two queues to hold undraw information
Queue 1,num

ResetList p()        ;loop through list initialising planets
While AddItem(p())
 \angle=Rnd(1000),Rnd(20)+50,Rnd(1)/30+.01,Rnd(15)
Wend

Gosub makeshapes     ;create 16 frames of animation

While Joyb(0)=0              ;while mousebuttons not down
 VWait:Show db:db=1-db:Use BitMap db ;flip display
 UnQueue db                  ;undraw aliens
 ResetList p()               ;loop through doing redraw and move
 While NextItem(p())
  QBlit db,\frame&15,160+Cos(\angle)*\dist,100+Sin(\angle)*\dist
  \angle+\speed
  \frame=\frame+\speed*20
 Wend
Wend

End

makeshapes:          ;subroutine to create 16 frames
 AutoCookie On       ;each shape needs a cookie cut
 Cls
 Circlef 20,20,16,6  ;draw shape in top left of bitmap
 Circle 20,20,16,2
 Boxf 2,18,38,22,2
 Boxf 0,19,40,21,4
 GetaShape 0,0,0,40,40   ;grab it off the bitmap
```

```
Cls                 ;clear the bitmap
MidHandle 0           ;position handle is at centre of shape
For i=1 To 15       ;generate other 15 shapes
 CopyShape 0,i        ;by copying original
 Rotate i,i/32      ;and rotating it
 MidHandle i          ;and setting its drawing handle
Next
Return
```

6. A conversion utility for Assembly Code to Blitz Assembler.

The following program is an example of a standard file conversion utility. It reads the file specified in the CLI argument and outputs a new file with some format revisions.

The input file is expected to be standard Assembler source code such as that generated by HiSoft's excellent DevPac editor/ assembler.

The **FileInput** and **FileOutput** commands redirect **Print**, **NPrint**, **Edit()** and **Edit$()** to work with the DOS files successfully opened by the **ReadFile** and **WriteFile** commands.

The **While Not EOF()..a$=Edit$(120)..Wend** structure loops through the input file a line at a time until the end of the file is reached.

Instances of "<<", ">>" and other strings are replaced by their Blitz 2 counterparts using the **Replace$** command.

I$, o$, p$ and r$ are temporary strings used to hold the different fields of a typical assembly code line.

The getw subroutine extracts the next file from the line and returns it in c$.

Once the various fields are filled and revised if necesary the print statement is used to output the new line to a file.

```
;
; convert genam2 type assembly source to blitz type assembly code
;
If NumPars<>1      ;make sure theres one parameter passed from the CLI
 NPrint "One parameter please darling"
 End
End If

CaseSense Off      ;dont worry about case sensitive for string searches

n$=Par$(1)                ;generate destination filename
n$=Replace$(n$,".s","")+".bb2"

If ReadFile(0,Par$(1)) AND WriteFile(1,n$) ;open both files
 FileInput 0:FileOutput 1            ;redirect input and output

 While NOT Eof(0)                    ;while still code to read
 ;
 a$=Edit$(128)                     ;get next line of code from file
 If Left$(a$,1)="*" Then a$=";"+a$  ;first character is a *? then make sure its a rem
```

```
a$=Replace$(a$,"<<","LSL")        ;replace <<
a$=Replace$(a$,">>","LSR")        ;and >>
a$=Replace$(a$,"endm","end macro") ;endm gets replaced by end macro
a$=Replace$(a$,"\","`")           ;macro parameters use key above tab instead of
;backslash
l$="":o$="":p$="":r$="":p=1       ;reset label$ opcode$ parameter$ and remark$
If Len(a$)=0 Then Goto dunline    ;check for null line

Gosub getw:If Len(g$)<>0          ;get the label field
 If Left$(g$,1)=";" Then r$=g$:Goto dunline
 l$=g$
 If l$<>""
  If Right$(l$,1)<>":" Then l$=l$+":"
  If Left$(l$,1)="." Then l$="'"+Mid$(l$,2)
 End If
End If
                                  ;get the opcode field
Gosub getw:If Len(g$)<>0
 If Left$(g$,1)=";" Then r$=g$:Goto dunline
 o$=g$
End If

Gosub getw:If Len(g$)<>0          ;get the parameter field (operands)
 If Left$(g$,1)=";" Then r$=g$:Goto dunline
 p$=g$
 p$=Replace$(p$,"(sp)","(a7)")
End If

g$=Mid$(a$,p)                     ;get the remarks field
If Len(g$)<>0
 r$=g$:If r$<>"" AND Left$(r$,1)<>";" Then r$=";"+r$
End If

dunline:

o$=LCase$(o$)                     ;make operand lower case

If o$="macro"                     ;macro moves back to label field
 o$=l$:l$="macro "                ;and name goes to operand field
End If

If o$="equ" OR o$="set"           ;#constant= instead of equ and set
 NPrint "#",StripTrail$(l$,Asc(":")),"=",p$," ",r$
 Goto popline
EndIf

If o$="section" OR o$="public" Then o$=";"+o$  ;rem out section and public operands


If l$<>"" Then NPrint l$          ;put label on its own line
If o$<>"" Then NPrint " ",o$," ",p$," ",r$ Else NPrint r$

popline:
;
Wend                              ;loop till end of file
```

```
CloseFile 0:CloseFile 1    ;close up

Else                       ;couldn't open both files
 NPrint "Unable to open file."
EndIf

End

;
;getw returns the next 'field' of characters
;
getw:
 If Mid$(a$,p,1)=";" Then g$=Mid$(a$,p):p=Len(a$):Return
 q$=Mid$(a$,p,1):If q$="'" OR q$=Chr$(34) Then Goto quotes
 g$=""
 gc:c$=Mid$(a$,p,1):If c$<>" "AND c$<>""AND c$<>Chr$(9) Then p+1:g$+c$:Goto gc
 gs:c$=Mid$(a$,p,1):If c$=" " OR c$=Chr$(9) Then p+1:Goto gs
 Return
quotes:
 g$=Chr$(34):p+1
 gq:c$=Mid$(a$,p,1):If c$<>q$ AND c$<>"" Then p+1:g$+c$:Goto gq
 g$+Chr$(34):p+1:Goto gs
```

7. A StarField Routine.

The following program is a simple fly through the starfield routine again using a list to keep track of the stars as well as reading the mouse in Blitz mode so that the user can control speed and rotation.

An 8 colour display is setup using the **BitMap**, **Slice** and **Show** commands as usual.

The **NewType** .star holds the polar coordinates of the star (angle and distance) as well as a speed, acceleration and the stars screen coordinates.

An array of type quick is used to hold a 'look up table' of sine and cosine values. This is much quicker than calling the floating point **Sin** and **Cos** routines within the main program loop and is accurate enough for our purposes. As there are 2*pi radians in a full circle (360 degrees) and we have 1024 entries in each look up table the coefficient i*pi/512 is needed to calculate the right radians value for each entry i. Note that **pi** is a physical constant in Blitz 2.

The main loop reads the mouse position using the **MouseX** and **MouseY** commands.

Looping through each item in the List the program moves each star, calculates it's screen coordinates by converting its polar coordinates to cartesian values, and plots it on the screen. If the star is outside screen coordinates it is removed from the list using the **KillItem** command.

Once all the stars in the list are drawn, the program attempts to add a new star to the list if there is any room.

```
;
;star field routine
;

num=128
```

```
BLITZ                ;setup 8 colour display
BitMap 0,320,256,3
Slice 0,44,3
Show 0
Mouse On             ;turn on blitz mouse driver
MouseArea 0,0,1024,256

NEWTYPE.star         ;setup stars
 angle.w             ;each star has 6 bits of information
 dist.q:speed:acc    ;
 sx.w:sy             ;including last screen position plotted
End NEWTYPE

Dim List stars.star(num)

USEPATH stars()

;the following sets up an array of quicks to hold sin and cos
;tables using 1024 to 2*pi ratio, this is very quick way for
;doing polar type geometry

Dim qsin.q(1023),qcos.q(1023)    ;setup qsin and qcos arrays
For i=0 To 1023
 qsin(i)=Sin(i*Pi/512):qcos(i)=Cos(i*Pi/512)
Next

For i=1 To 7     ;set up palette as grey scale so stars are
 br=4+i*1.7      ;brighter as they get closer
 RGB i,br,br,br
Next

While Joyb(0)=0  ;while mouse buttons not pressed

 mx=MouseX       ;read mouse for this frame
 my=MouseY/512

 ResetList stars()          ;move and draw stars
 While NextItem(stars())
  Plot \sx,\sy,0            ;rub out old plot
  \speed+\acc+my            ;add acceleration and mousey to star's speeed
  \dist+\speed              ;then add its speed to distance
  \sx=160+qcos((\angle+mx)&1023)*\dist     ;calculate new sx and sy
  \sy=128+qsin((\angle+mx)&1023)*\dist
  If \sx<0 OR \sx>319 OR \sy<0 OR \sy>256
   KillItem stars()                ;if reached screen borders delete from list
  Else
   Plot \sx,\sy,\dist/20           ;else plot it on screen
  EndIf
 Wend

 If AddItem(stars())    ;add a star if theres a space in the list
  \dist=0
  \speed=0
  \acc=Rnd(1)/32
  \angle=Rnd(1024)
 EndIf
```

```
Wend

End
```

8. List Processor for Exec based Lists using Pointer types

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 select the 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 nows 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
```