INFORMATION AND SUPPORT FOR BLITZ BASIC 2 USERS WORLD WIDE

BLITZ USER

ISSUE 2 November1992

YES its the one everybody has been waiting for, ISSUE 2 of Blitz USER magazine! We're still alive and kicking, don't worry about that! Featuring a whole host of new commands including DPaint anim support, ARexx commands for your Blitz applications, system friendly serial port commands, speech, MED sequencer and more. And this time its on a disk so you don't have to type them in! ALSO INCLUDES PRE-RELEASE OF BLITZ 3D, *get into it!*

HINTS TRICKS & TIPS

*Parameters always need to be in brackets when using Blitz 2 functions (commands that return a value).

*If you want your program to run from the workbench always use the WBStartup command at the very top of your program.

*Always have runtime errors enabled when testing your programs.

*Always disable runtime errors when testing your program for speed.

*Turnoff overflow errors in the runtime errors requester if you do not want your program stopped when var.b>127, var.w>32767 etc.

*Always select Make-Smallest code in the options requester when you are creating an executable file.

*Never return from a subroutine from within a Select..Case structure without doing a Pop Select before the Return.

*Use a SetErr:End:End SetErr to stop programs crashing with runtime errors disabled when an error occurs, if for instance the program is run on an Amiga with too little memory or from the wrong directory this will ensure a clean exit without a guru.

*Use shift-leftarrow to move the cursor across to the same indent as the line above when writing structured programs.

*Delete the l:BlitzEditor.opts file if you have changed from running your Amiga workbench in noninterlace/interlace. The editor will run in the same resolution as that of Workbench if it does not find the .opts file.

*If you accidentally loose some of your program and save it try loading the .bak file which contains your program as saved before the last save.

*If you want a command added to Blitz 2 write to Acid Software.

*Use BBlit not QBlit if your Blits are messing up the background.

*Don't use more than one condition in If Then structures with commands like OpenFile, ReadFile, AddItem i.e. dont use If data=1 And OpenFile(blah) as even if data<>1 the file will still be opened.

*Use the EVEN directive after dc.b if you want subsequent data to be word aligned.

*The ds instruction in Blitz does not put zeros in the area like Genam does so if converting machine code to Blitz replace with dcb.

*Always back up your programs on separate disks just to be safe.

*If your program operates on strings always make sure the string buffer setting in options is set to the largest possible string your program will deal with (default=10240). The other settings are all for compile time buffers.

*Never, ever, ever let your friends make photocopies of Blitz manuals and User magazines.

No.2

CONTENTS

Editorial 4
No we haven't forgotten about you, in fact we've packed this issue nice and full, just for you!

Letters 5
Public opinion at it's finest from Blitz 2 users world wide.

Back in the office... 6
Latest rumours about our wonderfuyl office.

Groovy Routines 7
Mark contributes cd, getcd{} and date{}

Buzz Bar 2 10
Part 2 of the game that wasn't too hot but is now!

IsoBlocks 13
A look at drawing in an isometric perspective.

NEW COMMANDS 15

Documentation to the update included on the cover disk.

AnimLib 15
Various commands 17
SpeakLib 20
MedLib 23
SerialLib 25
Arexx 28
3DLib 39

Library Development 43
More in depth docs for the professionals.

BLITZ USER

Blitz User is a publication of Acid Software.

Duplication of this magazine is prohibited however all ideas and programs included in this magazine may be used in any size, shape or form.

Acid Software takes no responsibility for the reliability of programs published in this magazine.

Editor

Simon Armstrong

Art Director

Rod Smith

Forward all contributions, advertising and correspondence to:

ACID SOFTWARE
10 StKevins
Arcade
Karangahape Rd
Auckland
New Zealand

fax:64-9-358-1658

EDITORIAL

Thanks to everyone who has written to us after buying Blitz 2. Sorry if we haven't replied, we're a small operation and hopefully we'll cover most of peoples complaints and suggestions in either this issue or the next of Blitz User, a magazine that's looking like another manual!

I'm really looking forward to getting Blitz-Net up and running, I spend far too much time on local boards arguing about how uncompetetive Amiga is with PC and how stupid the new chip set is and this and that.

I'd much prefer to be logging onto a board that concentrated on programming ideas and fantasies, looking after Blitz users and the like, so hopefully, very soon now in fact, we'll have something up and running.

The main nodes will be in Koln Germany, Arizona USA, Sydney Australia and of course Auckland New Zealand.

One of the main objectives will be to allow users to download entire week loads of mail quickly so as not to cost much phone time and use floppy-mail for inexpensive transfer of large programs.(In most places it costs the same to put a floppy disk in the mail as it does to log onto a BBS long distance for 2 minutes).

The main point is that not many people have much money left over after **buying** software.

O.K. the main objective of this issue of Blitz User is new commands that Mark and others have added to Blitz BASIC 2. A lot of these were in response to user's requests so don't say we ignore everyone. We at Acid Software want to continually upgrade Blitz every few months, its hopefully going to be one of our big selling points, and we're not going to charge the earth either!

So, if you want something added please write describing what it is exactly you want, how you think we can implement it and if possible some code (even in C) that does the job.

As for sales, they have been ticking along, there's advertisments coming out in both Amiga World and some German mags that should get things going. We do depend on word of mouth a lot as we can not yet afford massive advertising campaigns telling the world about the wonders of Blitz.

So, if you can spread the word, write some demos, the more we sell the better support we can afford to give you the user.

I expect to ship the 1000th copy by the end of November which will be bang on target.

Anyway, back to this issue of Blitz User, thanks to **Greg Abiss** for the Arexx library and documentation.

What with multimedia and all the awesome software coming out for the Amiga, ARexx is definately a bonus for prductivity. Now thanks to **Greg** you can run your Blitz programs from Arexx scripts as well as controlling other applications as well.

Mark has added support for anim brushes as well as full screen anims. There are quite a few formats for anim files, we decided that to start with we will just support that which DPaint supports, fair enough yes?

Thanks to **Paul Andrews** from **Vision Software** for providing the Med3 player, **Teijo Kinnunen**'s med format is pretty special (if you're interested, the MED editor is available from AmigaNuts United).

Since giving up trying to give up smoking i've been super productive, spent a few hours fixing up BuzzBar (which I hope to get on a proper magazine's cover disk soon) got this magazine finished, and have even managed to get some 3D working to dispel rumours that it is all a big hoax.

Enjoy, and don't expect another issue if you don't send us some friendly mail.

SIMON

LETTERS

The following section is for publishing any correspondence received at Acid Software in New Zealand. To write to us please use the following address:

Acid Software
10 St Kevins Arcade
183 Karangahape Road
Auckland
New Zealand

If you have a fax our number is:

64-9-358-1658

The leading 64 is the international area code for New Zealand.

Due to the serious nature of the mail we have received since the last issue we have decided to mix in a little humour of our own along with, for the first time, some genuine user correspondence!

Greetings to the producers of Blitz BASIC 2. 10 points for a fantastic BASIC language but I'm having some problems that you can clear up, Project BuzzBar fails at function.w angle ext.l d0 Why? What is the .bak file for? What do I need in the C, libs, l etc. dirs for a self booting disc for insectoids?

Cheers,

Yif Howell
Newtown, Australia

O.K. Yif. to answer your points in order...

1. Part 1 of Project BuzzBar is included on this months cover disk so you should be able to see where you went wrong from the files on the disk.

2. The .bak file is your program as it was the last time you saved it, this is a backup so that if you accidentally delete an important section of your program and save it you can still load the .bak file and recover the section you lost. Blitz renames your old file as the .bak and then saves the new version, this also means that if use a hard disk, the source code is likely to be written to a different part of the disk increasing the likely hood of recovery if your drive crashes, a likely event until Quantum kicked Seagates butt (hard disk talk).

3. I would think the easiest way would be to copy a PDDisk as they have all the necesary files. Do a search for any font descriptors in the Insectoid source and make sure you have included those fonts in the fonts drawer.

As for your other questions that we didn't print, I'll get round to them next issue.

Dear Simon,

Here are a few extra suggestions:

1. The file requester should be more customisable, a device selector option would be useful.
2. MAX and MIN functions.
3. A function that reads the system timer.
4. file access is awkward, it would be easier to use Print#.
5. A boolean primitive type.
6. A menu option to list all variables in your program.
7. MOD and DIV functions.
8. The function SPACE$
9. A data gadget

Yours Sincerely
Michael Green

Hi Michael, thought I'd print this so others can go hmmm yes that would be nice, we are expanding the gadget library and a simple debugger will give you variable lists etc. Should be able to tackle some of this list in the next issue.
Simon

Hi guys,

Thanks for a really amazing language. Using Blitz I can gradually work into assembler, I'm glad that Blitz is as astounding as it is, cos I've had bad (well, really slow) experiences with certain other hyped languages. OK now for a bug report...

1. demo disk source files are a bitbuggy.
2. I need a command to return the pixelwidth of a proportional text string and a style command.
3. Data such as shapes can

not be saved with the executable.

8. A built in data compactor would be nice ala PowerPacker.

9. Blitz gurus sometimes when the wrong disk is inserted

10. Can someone do an article on the copper and advanced effects...

Thanks for such a powerful Amiga-friendly system, unlike AMOS which whacks 50K onto any executable file!

Thanks Again
Damian Caynes
Murwillumbah
Australia

p.s How about Blitz contacts? so users worldwide can contact each other!

Yo Damian, well we don't want everything do we? Can we send you a free Amiga 4000 as well?

No seriously yes, yes, and yes we're looking into it.

Hey BlitzMan,

Just wondering if you would mind featuring in a game I'm writing, it will be lots of fun.

Mario

p.s. Promise you will never get killed!

Hi Simon,

Just thought I'd let you know that this is not some complicated BBS so there is no excuse for sending yourself mail!

Regards,
Simon

DISCLAIMER: The views, attitudes and ideas expressed in this section are not necesarily in agreement with the editors of this

BACK IN THE OFFICE...

Vision Software, developers of the game you wouldn't let your kids watch you play Zombie Apocalypse and the machine code shareware masterpieces Microbes and Cybernetix have nearly completed what is certain to be a top 10 smash hit called **Woody's World**. Keep your eye out for this New Zealand blockbuster in the New Year.

Mark Sibly is currently developing a sequel to the great defender game, this version however will not be a Blitz PD game as he's gone back to his favourite assembler, **Devpac 2**, to come up with what could be the most addictive action yet to come from outside the William's stables.

On the same track the Acid Software Stargate machine's current high score still stands at 1,315,875 (Mark of course).

High scores for Blitz 2 PD games will not be given great attention due to rumoured tampering of source code before users clocked their reported scores.

The famous rock band **Too Much Too Soon** featuring Mark Sibly on lead, Rod on rhythm, Simon on bass and two friends has disbanded. The office now has a little more room for computers now the drum kit and wall to wall Marshalls are no longer in residence.

Office nicotine levels are at an all time high now Simon is back chain smoking with the rest of the office, 200g of coffee is still being drunk per week, most popular CD of the month goes to the new Suicidal Tendency's disk the Art of Rebellion.

Office personalities will be interviewed in Blitz User next issue so if you have any questions, write in.

GROOVY ROUTINES

Welcome to a new Blitz User regular feature. Each month, this column will present a series of useful little routines you may wish to use in your own programs. All routines have been written as 'self-contained' as possible, allowing you to save them to disk for later 'include'-ing.

Many of these routines make use of Amiga library calls. You can identify library calls through the '_' suffix on commands - for example '**Lock_**'.

Note that when you are using Amiga library calls, they may be either in the form of functions (ie - they return a result) or in the form of statements (ie - any value returned is ignored). If you are using a library call as a function, you MUST enclose any parameters with brackets.

For example:

```
l.l=Lock_ "myfile",-2
```

is NOT legal, and will cause a syntax error at compile time.

```
l.l=Lock_("myfile",-2)
```

is the correct way to do this.

Likewise, library calls of the statement form should not normally have their parameters bracket enclosed.

For example:

```
Examine_(cd,ex)
```

is NOT legal, and will also cause a syntax error at compile time.

```
Examine_ cd,ex
```

is the correct way to do this.

Another thing to be wary of when using any of these routines is global variable conflicts. For example, the 'DATE.BB2' routines make use of 3 global arrays ('mnth()', 'mon$()', 'day$()'), and one global variable (thedate). When using this routine, you must be careful that these variable names are not used elsewhere.

And now, let's get on with this month's groovy routines...

GROOVY ROUTINE #1

This useful little function allows you to set the current directory from within your programs. This is similar to typing 'cd pathname' from the cli.

The cd function takes one parameter, the directory you wish to change to, and returns a true (-1) value if the directory change was successful, or a false (0) value if unsuccessful. For example:

```
XINCLUDE "cd.bb2"

If cd{"work:"}
      ;
      ;cd successful!
      ;
Else
      ;
      ;cd unsuccessful...
      ;
EndIf
```

```
;**********START OF CD.BB2********
Function cd{n$}
;
l.l=Lock_(&n$,-2)
If l
     CurrentDir_ l
     Function Return -1
Else
     Function Return 0
EndIf
;
End Function
;
;********* END OF CD.BB2 **********
```

GROOVY ROUTINE #2

This function may be used to determine the current directory. This is similar to typing 'cd' at the cli with no parameters.

Here is an example of using getcd:

```
XINCLUDE "getcd.bb2"
a$=getcd{}
NPrint a$
MouseWait


;*******START OF GETCD.BB2 ********
;
Function$ getcd{} ;return text of current directory...
;
;allocate memory for a fileinfo block...
ex.l=AllocMem_(260,1)
;
;this bit of magic picks up the current
;directory lock!
cd.l=Peek.l(Peek.l(Peek.l(4)+276)+152)
;
;make a duplicate of this lock so we
;can safely unlock it!
cd=DupLock_(cd)
;
Repeat
      ;
      ;fill in fileinfo block
      Examine_ cd,ex
      ;
      ;pick up directory name
      n$=Peek$(ex+8)
      ;
      ;go to parent dir & unlock old dir
      cd2.l=cd
      cd=ParentDir_(cd):UnLock_ cd2
      ;
      ;if no parent, then this is root.
      If cd=0 Then n$+":" Else n$+"/"
      ;
      ;add name to curent dir name
      cd$=n$+cd$
      ;
Until cd=0
;until no more parent directories
;
;free fileinfo block mem
FreeMem_ ex,260
;
Function Return cd$
;
End Function
;
;********* END OF GETCD.BB2 *********
```

GROOVY ROUTINE 3

By including this bit of code in your programs, you can easily determine the date and time.

The 'date{}', 'date2{}' and 'time{}' functions all return strings. the date functions return strings reflecting the current date, in 2 different formats, and the time function returns the time.

Before using these functions, you should first use the 'getdate{}' statement. This reads the current date and time into a 'thedate' variable. The date and time functions simply decode this information into appropriate strings.

Here is an example:

```
XINCLUDE "date.bb2"
getdate{}
NPrint date{}
NPrint date2{}
NPrint time{}
Mousewait


;******** START OF DATE.BB2 *************

;
NEWTYPE.dateinfo
;
d1.l ;day count
d2.l ;minutes count
d3.l ;ticks count (50 ticks per second)
;
year.w
month.w
day.w
hours.w
mins.w
secs.w
;
End NEWTYPE

DEFTYPE.dateinfo thedate
USEPATH thedate

Dim mnth(12),mon$(12),day$(7)

mnth(1)=31:mon$(1)="January"
mnth(2)=28:mon$(2)="February"
mnth(3)=31:mon$(3)="March"
mnth(4)=30:mon$(4)="April"
mnth(5)=31:mon$(5)="May"
mnth(6)=30:mon$(6)="June"
mnth(7)=31:mon$(7)="July"
mnth(8)=31:mon$(8)="August"
mnth(9)=30:mon$(9)="September"
mnth(10)=31:mon$(10)="October"
mnth(11)=30:mon$(11)="November"
mnth(12)=31:mon$(12)="December"
```

```
day$(1)="Sunday"
day$(2)="Monday"
day$(3)="Tuesday"
day$(4)="Wednesday"
day$(5)="Thursday"
day$(6)="Friday"
day$(7)="Saturday"

Statement getdate{}
Shared thedate,mnth()
DateStamp_ &thedate
y=1978
d=\d1+1
While d>365
     d-365
     If y MOD 4=0 Then d-1
     y+1
Wend
m=1
mnth(2)=28:If y MOD 4=0 Then mnth(2)=29
While d>mnth(m)
     d-mnth(m)
     m+1
Wend
\year=y
\month=m
\day=d
\hours=\d2/60
\mins=\d2 MOD 60
\secs=\d3/50
End Statement

Function$ num{n}
Function Return Right$("0"+Str$(n),2)
End Function

Function$ date{}
Shared thedate,mon$(),day$()
d$=day$(\d1 MOD 7+1)+" " +Str$(\day)+ " " +mon$(\month)+" "+Str$(\year)
Function Return d$
End Function

Function$ date2{}
Shared thedate
Function Return num{\day}+"/"+num{\month}+"/"+Str$(\year)
End Function

Function$ time{}
Shared thedate
Function Return num{\hours}+":"+num{\mins}+":"+num{\secs}
End Function
;
;*************** END OF DATE.BB2 *************
```

BUZZ BAR 2

Apologies are in order for doing such a rush job in the first issue ala the scribble at the bottom of the listing. My accelerated machine failed to inform me that buzzbar was infact running as slowly as an Amos program... oops I thought.

The number 50 that needed changing was located in drawstars NOT setupstars, so before you go any further go to the version you typed in and try changing it, better?

The new version has the source code on the disk ("thank goodness" I hear you say), I have printed it here so you can refer to the following explanations as we go.

First I have introduced a dual playfield display, with a nice big backdrop that scrolls. The IFF is 512x512 but our bitmap is 832x768 so it can wrap around invisibly. Without being complicated, the scroll command copies the left edge of the bitmap to the right so once we have scrolled across to the right we can reset the scroll counter to 0 without the display changing. See the examples disk for simpler code examples of dual playfields and scrolling.

The ssin and ccos macros just saved some typing, !ssin{var} is expanded by the compiler into:

qsin((var **lsr** 6)&1023)

As you may have noticed this is actually only required because Blitz BASIC's **lsr** doesn't work quite right!

The onscreen macro checks if the

```
INCLUDE qfuncs.bb

NEWTYPE .ship
        x.w:y:rot:thrust:rspeed:id:frame:xv:yv:px:py:upd
End NEWTYPE

Dim List nme.ship(50) ;enemy
Dim List bul.ship(50)  ;bullets
Dim List bng.ship(50) ;explosions

Dim qsin.q(1023)       ;look up tables
Dim qcos.q(1023)

me.ship\x=0,0,0,0,0,0,0

BitMap 0,320+64,256+64,3;doublebuffered display
BitMap 1,320+64,256+64,3
BitMap 2,832,768,3  ;background

LoadShapes 0,"ships.shapes"
LoadShapes 64,"bombs.shapes"
LoadShapes 80,"shards.shapes"

LoadBitMap 2,"moon.iff"
Use BitMap 2
Scroll 0,0,320,512,512,0
Scroll 0,0,832,256,0,512
Use BitMap 0

LoadPalette 0,"ships.iff"
LoadPalette 0,"moon.iff",8

Queue 0,100
Queue 1,100

BLITZ                       ;setup blitz display
Mouse On
BlitzKeys On:BitMapInput
Slice 0,44,320,256,$fffa,6,8,32,320+64,832
Use Palette 0

Gosub setupsincos
Gosub setupnme
Gosub setupdisplay

While NOT RawStatus($45) ;main loop
        VWait
        ShowF db,32,32
        ShowB 2,(me\x LSR 6)&511,(me\y LSR 6)&511
        db=1-db
        Use BitMap db
        UnQueue db
        Gosub drawnme
        Gosub drawbullets
        Gosub moveship
        Gosub moveexplosions
;       MOVE #$fff,$dff180
Wend

End

.setupdisplay
        Return

Macro ssin qsin((`1 LSR 6)&1023):End Macro
Macro ccos qcos((`1 LSR 6)&1023):End Macro
Macro onscreen RectsHit(`1,`2,1,1,12,12,320+32,256+32):End
Macro

#xcntr=160+32
#ycntr=100+32
```

x,y coordinates are inside the play area. Using macros like this is faster than calling procedures and keeps the program nice and tidy.

The moveship routine is the same except for one bug which is important. The old line

If AddItem(bul()) **And** rl=0

will add an item to the list even if rl<>0, this was leaving random bullets as a bullet item would be allocated even though it wasn't used. This is also applicable to functions like **OpenFile**() and users should be aware.

The move explosions routine is new, the explosion's rot value decrements to -1 drawing the animations of the shard. It is then removed from the list. The shards would look better if they looked as though they were falling towards the planet but my artistic skill just wasn't up to it.

The drawnme has a few additions. The \upd field is used to update the aliens direction every 6 frames, on initialisation the aliens are assigned a random \upd so they don't all get changed on the same frame. This is an example of speeding things up a lot with out any noticable change to the game play. They still get moved every frame.

The \frame field allows us to assign different shapes to different aliens.

Also the way I was moving them was pretty stupid, now instead of just rotating them a constant amount towards you, I treat \rspeed as a maximum amount of turn, this makes things much cooler!

Drawbullets now has collision detection. We first check to see if there is anything on the bitmap with BlitColl (see new commands) and if there is we search through each nme() to find who we have killed.

Easy huh! We then kill the nme with, yup, **KillItem** which removes it from the nme() list and add 5 shards to the explosion list, moving at random directions.

```
.moveship
    If RawStatus($31) OR Joyx(1)=-1 Then me\rot-1400
    If RawStatus($32) OR Joyx(1)=1 Then me\rot+1400
    me\rot+(MouseXSpeed*200)
    If RawStatus($38) OR Joyb(0)&2 OR Joyy(1)<>0;thrust
        me\xv+!ssin{me\rot} ASL 4
        me\yv-!ccos{me\rot} ASL 4
    EndIf
    If RawStatus($39) OR Joyb(0)&1 OR Joyb(1)&1;fire
        If rl=0
            If AddItem(bul())
                bul()\x=me\x,me\y,me\rot
                bul()\xv=!ssin{me\rot} ASL 8+me\xv
                bul()\yv=-!ccos{me\rot} ASL 8+me\yv
                rl=8
                me\xv-!ssin{me\rot} ASL 5 ;reverse thrust
                me\yv+!ccos{me\rot} ASL 5
            EndIf
        EndIf
    Else
        rl=0
    EndIf
    me\xv-me\xv ASR 5 ;drag
    me\yv-me\yv ASR 5
    me\x+me\xv
    me\y+me\yv
    QBlit db,((me\rot+2048) LSR 12)&15,#xcntr,#ycntr
    If rl>0 Then rl-1
    Return

.moveexplosions
    ResetList bng()
    USEPATH bng()
    While NextItem(bng())
        \rot-1
        If \rot>-1
            \x+\xv:\y+\yv
            px=((\x-me\x) ASR 6)+#xcntr
            py=((\y-me\y) ASR 6)+#ycntr
            If !onscreen{px,py}
                QBlit db,80+\rot/4,px,py
            EndIf
        Else
            KillItem bng()
        EndIf
    Wend
    Return

.drawnme
    ResetList nme()
    USEPATH nme()
    While NextItem(nme())
        \upd-1
        If \upd<0
            \upd=5
            ang.w=32768-angle{me\x-\x,me\y-\y}-\rot
            s=Sgn(ang):ang=Abs(ang)
            If ang>\rspeed Then ang=\rspeed
            \rot+s*ang ;rotate towards me
            \xv+!ssin{\rot} * \thrust;thrust
            \yv-!ccos{\rot} * \thrust
            \xv-\xv ASR 6;drag
            \yv-\yv ASR 6
        EndIf
        \x+\xv     ;speed
        \y+\yv
        \px=((\x-me\x) ASR 6)+#xcntr
        \py=((\y-me\y) ASR 6)+#ycntr
        If !onscreen{\px,\py}
            QBlit db,(\rot LSR 12)&15+\frame,\px,\py
        EndIf
    Wend
```

Last but not least the setupnme now selects an id for the 2 wonderful shapes I have drawn. Each has a different range of turning and thrust variables.

Things to do:

I've been a bit brief on the explanations front, if you haven't had much experience with writing arcade games the first thing to do is probably mess with the graphics.

The planet surface is called moon.iff load it into DPaint, change the palette, add some mountains, ravines, or go for a completely different look. Keep the colors dark so the foreground ships stand out.

If you want to change or add aliens, use DPaint to mess with the ships.iff picture. Once finished, load the new page into shapesmaker, enable auto-centre and create a ships.shapes file.

To change the difficulty of the aliens goto the setupnme and alter the two numbers in the \thrust=32,16384, these represent how fast the aliens can go and how fast they can turn towards you. Making them bigger will make them faster to react to your change in position.

Well thats it for this issue, as with the other games on the disk you can only release playable versions if you add sound effects and put a Blitz 2 reference somewhere IN THE GAME.

Look forward to seeing some new graphics at least!

Simon

p.s.

Use the escape key to exit from the game, next issue we probably want attack waves, a front end, 5 lives and a null modem setup as promised last issue!

```
      Return

.drawbullets
      ResetList bul()
      USEPATH bul()
      While NextItem(bul())
            \x+\xv     ;speed
            \y+\yv
            px=((\x-me\x) ASR 6)+#xcntr
            py=((\y-me\y) ASR 6)+#ycntr
            If !onscreen[px,py]
                  If BlitColl(64,px,py)
                        ResetList nme()
                        While NextItem(nme())
                        If
ShapesHit(16,nme()\px,nme()\py,64,px,py)
                        For i=1 To 5
                        If AddItem(bng())
                        bng()\x=\x,\y,31
                        bng()\xv=nme()\xv/2+Rnd(64)-32
                        bng()\yv=nme()\yv/2+Rnd(64)-32
                        Endif
                        Next
                        KillItem nme()
                         Endif
                        Wend
                  Endif
                  QBlit db,64,px,py
            Else
                  KillItem bul()
            Endif
      Wend
      Return

.setupships:
      For i=1 To 15
            CopyShape 0,i;generate rotations
            Rotate i,i/16
            MidHandle i
      Next
      Return

.setupnme:
      USEPATH nme()
      For i=1 To 16
            AddItem nme()
            \x=Rnd(65535),Rnd(65535),Rnd(65535)
            \id=Rnd(2)
            Select \id
                  Case 0:\thrust=32,16384:\frame=16
                  Case 1:\thrust=20,8192:\frame=32
            End Select
            \upd=Rnd(6)
      Next
      Return

.setupsincos
      For i=0 To 1023
            r.f=i*Pi/512
            qsin(i)=Sin(r)
            qcos(i)=Cos(r)
      Next
      Return
```

ISO-BLOCKS

ISO-BLOCKS is an example of using isometric perspective to display three dimensional graphics.

It is a useful method for games as well as displaying 3D bar graphs.

The illustration on the following page is a basic diagram of how the x,y and z axis are displayed on the screen.

The file blocks.iff and blocks.shapes in the iso_blocks drawer of the cover disk contain blocks and spheres in 5 colours that the listing loads in the first two lines.

The program then opens a hires interlace screen, sets the palette correctly and defines the screensbitmap as BitMap#0.

The macro !p converts an x,y,z coordinate to a screen x and y coordinate.

'1 '2 and '3 are replaced by the x,y,z parameters when the macro is called.

This means when we reference the macro in the drawgrid routine:

Blit c-1,!p{x,y,z}

the compiler expands the macro so the line reads...

```
LoadShapes 0,"blocks.shapes"
LoadPalette 0,"blocks.iff"

Screen 0,16+8+4
ScreensBitMap 0,0
Cls:Use Palette 0

#xoff=320
#yoff=252

Macro p #xoff+`1*9-`2*6,#yoff+`1*3+`2*6-`3*7:End Macro

Dim grid.b(30,30,30)

Statement gplot{x,y,z}
 Shared grid(),pen
 grid(x+15,y+15,z+15)=pen
End Statement

Statement cube{x,y,z,w,l,h}
 For zz=z-h To z+h
  For yy=y-l To y+l
   For xx=x-w To x+w
    gplot{xx,yy,zz}
   Next
  Next
 Next
End Statement

Statement drawgrid{}
 Shared grid()
 For z=-15 To 15
  For y=-15 To 15
   For x=-15 To 15
    c=grid(x+15,y+15,z+15)
    If c Then Blit c-1,!p{x,y,z}
   Next
  Next
 Next
End Statement

For i=14 To 0 Step -1        ;draw pyramid
 pen=(i MOD 4)+5
 cube{0,0,-i,i,i,0}
Next

drawgrid{}

MouseWait

End
```

ISO-Blocks listing

Blit c-1,#xoff+x*9-y*7,#yoff+x*3+y*6-z*7

The grid(30,30,30) array represents each point in a three dimensional space similar to a 3D checkers game, grid(15,15,15) represents the very middle of this space.

The gplot statement sets the appropriate point in the grid space to the value of pen.

The cube statement colours in a cube of points in the space starting at point x,y,z with w=width, l=length and h=height.

The drawgrid{} statement draws all the points in the grid array as blocks on the screen with the **Blit** command.

The blocks have to be drawn from furthest to nearest so that blocks behind others never appear on the screen as being in front of them.

The pyramid loop draws squares starting at the base getting smaller as it loops upwards.

Things to do:

OK, thats the concept in place. I've had a few ideas for expanding on this program.

First to get a better idea of the space you're working in insert a temporary line after c=grid(.. in the drawgrid{} statement to c=int(rnd(5))+1 to fill the screen with blocks disregarding the contents of grid().

Use c=int(rnd(5))+6 to draw spheres.

Now take out the 4 lines that draw the pyramid and try drawing other paterns using the cube statements such as...

```
pen=1:cube{0,0,0,15,1,1}
pen=2:cube{0,0,0,1,15,1}
pen=3:cube{0,0,0,1,1,15}
pen=6:cube{0,0,0,3,3,3}
```

If you wish, you could change the blocks in the iff so that there are lines on the top and sides of adjacent bricks not just on the fronts. I think this would improve their look a lot.

Use shapesmaker to generate the new .shapes file (set auto centre before converting them).

You could also try desigining a 32x32 pixel 32 colour bitmap in dpaint and use the colour of each pixel to draw a vertical bar on the grid with height corresponding to the pixel colour.

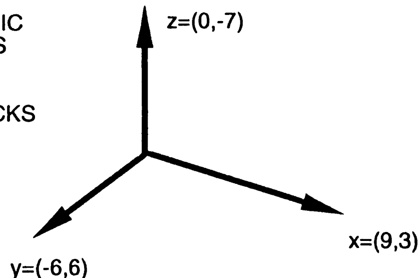
The code would look something like the following, don't forget to load the iff into bitmap 1 and maybe change pen colour for each bar.

```
;32x32 picture in bitmap 1
use bitmap 1
for x=-15 to 15
  for y=-15 to 15
    ;get colour of pixel
    c=point(x+15,y+15)
    ;draw a vertical bar
    cube{x,y,-15,0,0,c}
  next
next
use bitmap 0
```

Then I got to thinking that a 3D logo program would be pretty cool, a zero gravity turtle! Not only would it move and turn normally but could step up and down "levels", however I got a bit stumped, someone might be able to get something going yes?

This program might have also given you a few ideas about how to do a crystal castles (Atari) type game perhaps, or a 3Dcheckers, or a chart generator. If you want to add menus don't forget to open a backdrop, borderless, nogadget full size window after opening the screen from which to hang your menus.

Simon.



NEW COMMANDS

ANIM.LIB

The following 4 commands allow the display of Animations in Blitz BASIC. The Animation must be compatible with the DPaint 3 format, this method uses long delta (type 2) compression and does not include any palette changes.

Anims in nature use a double buffered display, with the addition of the ShowBitMap command to Blitz we can now display (play) Anims in both Blitz and Amiga modes. An Anim consists of an initial frame which needs to be displayed (rendered) using the InitAnim command, subsequent frames are then played by using the NextFrame command. The Frames() function returns the number of frames of an Anim.

We have also extended the LoadShape command to support Anim brushes.

The following example loads and plays an Anim on a standard Amiga (Intuition) Screen.

```
;
;play anim example
;

;anim file name could use f$=par$(1) to play anim from cli

f$="test.anim"

;open screen same resolution as animation

ILBMInfo f$
Screen 0,0,0,ILBMWidth,ILBMHeight,ILBMDepth,ILBMViewMode,"",1,2
ScreensBitMap 0,0

;open extra bitmap same size as screensbitmap for double buffering

BitMap 1,ILBMWidth,ILBMHeight,ILBMDepth

;load anim and set screen colours to same as animation

LoadAnim 0,f$,0
Use Palette 0

;draws first frame to current bitmap (1) and bitmap #0

InitAnim 0,0

While Joyb(0)=0
    ShowBitMap db       ;tell intuition which bitmap to display
    VWait               ;wait for top of frame
    db=1-db             ;swap current bitmap
    Use BitMap db
    NextFrame 0         ;and draw next frame
Wend
```

Statement: **LoadAnim**

Syntax: **LoadAnim** *Anim#,FileName$[,Palette#]*

Modes: Amiga

Description:

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

The optional *Palette#* parameter can be used to load a palette with the anims correct colours.

Notes: unlike more advanced anim formats DPaint anims use a single static palette for the entire animation. Like all other Blitz commands that access files the command must be executed in Amiga mode.

Statement: **InitAnim**

Syntax: **InitAnim** *Anim#[,Bitmap#]*

Modes: Amiga/Blitz

Description:

InitAnim renders the first two frames of the Anim onto the current BitMap and the *BitMap* specified by the second parameter. The second *BitMap#* parameter is optional, this is to support Anims that are not in a double-buffered format (each frame is a delta of the last frame not from two frames ago). However, the two parameter double buffered form of InitAnim should always be used. (hmmm don't ask me O.K.!)

Statement: **NextFrame**

Syntax: **NextFrame** *Anim#*

Modes: Amiga/Blitz

Description:

NextFrame renders the nextframe of an Anim to the current BitMap. If the last frame of an Anim has been rendered **NextFrame** will loop back to the start of the Animation.

Function: **Frames**

Syntax: **Frames** *(Anim#)*

Modes: Amiga/Blitz

Description::

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

VARIOUS NEW COMMANDS

Statement: **ShowBitMap**

Syntax: **ShowBitMap** *[BitMap#]*

Modes:Amiga

Library: ScreensLib

Description:

The **ShowBitMap** command is the Amiga-mode version of the **Show** command. It enables you to change a Screens bitmap allowing double buffered (flicker free) animation to happen on a standard Intuition Screen.

Unlike Blitz mode it is better to do ShowBitMap then VWait to sync up with the Amiga's display, this will make sure the new bitmap is being displayed before modifying the previous BitMap.

Function: **BlitColl**

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

Modes: Amiga/Blitz

Description:

BlitColl is a fast way of collision detection when blitting shapes. **BlitColl** returns -1 if a collision occurs, 0 if no collision. A collision occurs if any pixel on the current BitMap is non zero where your shape would have been blitted.

ShapesHit is faster but less accurate as it checks only the rectangular area of each shape, where as **BlitColl** takes into account the shape of the shape and of course1bcan not tell you what shapeyou have collided with.

Note: make sure only things that you want to collide with have been drawn on the BitMap e.g. don't Blit your ship and then try **BlitColl**!

Statement: **ILBMViewMode**

Syntax: **ILBMViewMode**

Modes: Amiga/Blitz

Library: ILBMIFFLib

Description:

ILBMViewMode returns the viewmode of the file that was processed by **ILBMInfo**. This is useful for opening a screen in the right mode before using LoadScreen etc.

The different values of ViewMode are as follows (add/or them for different combinations):

32768 ($8000) hires
2048 ($0800) ham
128 ($0080) halfbright
4 ($0004) interlace
0 ($0000) lores

See Also: ILBMInfo

Example:

```
;
;ilbminfo example
;

;iff file name could use f$=par$(1) to use cli argument

f$="test.iff"

;get ilbm information
ILBMInfo f$

;open screen with correct parameters
Screen 0,0,0,ILBMWidth,ILBMHeight,ILBMDepth,ILBMViewMode,"",1,2

;load the iff onto the screens
LoadScreen 0,f$,0

;set the palette
Use Palette 0

MouseWait
```

Statement: **LoadShape**

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

Modes: Amiga

Description:

The **LoadShape** command has now been extended to support anim brushes, if the file is an anim brush the shapes are loaded into consecutive shapes starting with the Shape# provided.

Statement: **ReMap**

Syntax: **ReMap** *colour#0,colour#1[,Bitmap]*

Modes: Amiga/Blitz

Library: Sis2dLib

Description:

ReMap is used to change all the pixels on a BitMap in one colour to another colour. The optional BitMap parameter will copy all the pixels in Colour#0 to their new colour on the new bitmap.

Statement: **ShapeGadget**

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

Mode: Amiga

Description:

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

The **ShapeGadget** command has been extended to allow an alternative image to be displayed when the gadget is selected.

All other parameters are identical to those in **TextGadget**.

Example:

```
;
;ShapeGadget example
;

Screen 0,3
ScreensBitMap 0,0

;generate 2 shapes for our shape gadget

Cls:Circlef 15,15,15,2:Circlef 8,8,9,5,3:Circlef 24,8,9,2,3
GetaShape 1,0,0,32,32:Circlef 24,8,9,5,3:GetaShape 0,0,0,32,32

ShapeGadget 0,148,50,0,1,0,1
TextGadget 0,140,180,0,2,"EXIT"
Window 0,0,0,320,200,$100f,"ClickMe",1,2,0

Repeat
Until WaitEvent=64 AND GadgetHit=2
```

Statement: **SetBPLCON0**

Syntax: SetBPLCON0 Default

Modes: Amiga/Blitz

Description:

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

bit#1-ERSY external sync (for genlock enabling)
bit#2-LACE interlace mode
bit#3-LPEN light pen enable

Example:

```
;
; Blitz Interlaced Slice Example using BPLCON0
;
```

```
BitMap 0,640,512,4

; use SetBPLCON0 4 to set the lace bit on when slice is created

SetBPLCON0 4                    ;set lace bit

BLITZ

;bitmap width=1280 means slice's bitmap
;modulos miss each 2nd line

Slice 0,44,640,256,$fffb,4,8,8,1280,1280   ;cludge the modulo

;every vertical blank either show odd lines or even lines
;depending on the long frame bit of VPOSR hardware register

SetInt 5
    If Peek($dff004)<0 Show 0,0,0 Else Show 0,0,1
End SetInt

;draw lines to prove it

For i=1 To 1000
    Line Rnd(640),Rnd(512),Rnd(640),Rnd(512),Rnd(16)
Next

MouseWait
```

Speak Commands

The Amiga speech synthesiser can be activated using the following commands. The narrator.device has been upgraded in Workbench2.0 increasing the quality of the speech. With a bit of messing around you can have a lot of fun with the Amiga's 'voice', Also note that these are compatible with the commands used in BlitzUser1's speech program.

Statement: **Speak**

Syntax: **Speak** *string$*

Modes: Amiga

Description:

The **Speak** command will first convert the given string to phonetics and then pass it to the Narrator.Device. Depending on the settings of the Narrator device (see **SetVoice**) the Amiga will "speak" the string you have sent in the familiar Amiga synthetic voice.

Example:

```
NPrint "Type something and hit return..."
NPrint "(just return to exit)"
Repeat
  a$=Edit$(80)
  Speak a$
Until a$=""
```

Statement: **SetVoice**

Syntax: **SetVoice** *rate,pitch,expression,sex,volume,frequency*

Modes: Amiga

Description:

SetVoice enables you to alter the sound of the Amiga's speech synthsiser by changing:

rate: measured in words per minute. Default 150, range 40-400.
pitch: the BaseLine pitch in Hz. Default 110, range 65-320
expression: 0=robot 1=natural 2=manual
sex: 0=male 1=female
volume: 0 to 64
frequency: samples per second (22200)

As the following example shows you could very well rename the **Speak** command the Sing command!

Example:

```
;
; sing the praises of Blitz BASIC!
;
While Joyb(0)=0
    pitch=65+Rnd(255)
    rate=100+Rnd(200)
    SetVoice rate,pitch,1,1,64,22200
    Speak "BLITZ BASIC"
Wend
```

Function: **Translate$**

Syntax: **Translate$***(string$)*

Modes: Amiga

Description:

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

Example:

```
Print "Enter a Sentence ":a$=Edit$(80)
NPrint "Phonetic=",Translate$(a$)
MouseWait
```

Statement: **PhoneticSpeak**

Syntax: **PhoneticSpeak** *phonetic$*

Modes: Amiga

Description:

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

Function: VoiceLoc

Syntax: **VoiceLoc**

Modes: Amiga

Description:

VoiceLoc returns a pointer to the internal variables in the speech synthesiser that enable the user to access new parameters added to the V37 Narrator Device. Formants as referred to in the descriptions are the major vocal tracts and are separated into the parts of speech that produce the bass, medium and trebly sounds.

The new paramters are as listed

\flags: set to 1 if using extended commands
\f0enthusiasm: amount of pitch difference on accents default=32
\f0perturb: amount of "wurble" ie random shake default=0
\f1adj,\f2adj,\f3adj: pitch adjust for low medium and high frequency formants 0=default
\a1adj,\a2adj,\a3adj: amplitude adjust for low medium and high frequency formants 0=default
\articulate: speed of articulation 100=default
\centralize: amount of the centphon vowel in other vowels 0=default
\centphon: a vowel to which all others are adjusted by the \centralize: variable,
(limited to IY,IH,EH,AE,AA,AH,AO,OW,UH,ER and UW)
\AVbias,\AFbias: amount of bias added to voiced and unvoiced speech sounds, (y,r,w,m vs st,sh,f). **\priority**: task priority when speaking 100=default

Example:

```
;
; voiceloc() example
;

NEWTYPE .voicepars          ;new V37 parameters available
    flags.b
    f0enthusiasm:f0perturb
    f1adj:f2adj:f3adj
    a1adj:a2adj:a3adj
    articulate:centralize:centphon$
    avbias.b:afbias:priority:pad1
End NEWTYPE

*v.voicepars=VoiceLoc

*v\flags=1
*v\f0enthusiasm=82,90 ;old aged highly excited voice
*v\f1adj=0,0,0    ;these are fun to mess with
*v\a1adj=0,0,0
*v\centralize=50,"AO" ;no effect
*v\articulate=90
*v\avbias=20,20

Speak "COME ON EVERYBODY, DANCE? boom boom you like my body yes!"

End
```

MEDLIB

Statement: **LoadMedModule**

Syntax: **LoadMedModule** *MedModule# Name*

Modes: Amiga

Description:

The **LoadMedModule** command loads any version 4 channel Octamed module. The following routines support upto and including version 3 of the Amiganut's Med standard.

The number of MedModules loaded in memory at one time is only limited by the MedModules maximum set in the Blitz2 Options requester.

Like any Blitz commands that access files LoadMedModule can only be used in AmigaMode.

Statement: **StartMedModule**

Syntax: **StartMedModule** *MedModule#*

Modes: Amiga/Blitz

Description:

StartMedModule is responsible for initialising the module including linking after it is loaded from disk using the LoadMedModule command. It can also be used to restart a module from the beginning.

Statement: **PlayMed**

Syntax: **PlayMed**

Modes: Amiga/Blitz

Description:

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

Statement: **StopMed**

Syntax: **StopMed**

Modes: Amiga/Blitz

Description:

StopMed will cause any med module to stop playing. This not only means that **PlayMed** will have no affect until the next **StartMedModule** but silences the audio channels so they are not left

ringing as is the effect when PlayMed is not called every vertical blank.

Statement: **JumpMed**

Syntax: **JumpMed** *Pattern#*

Modes: Amiga/Blitz

Description:

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

Statement: **SetMedVolume**

Syntax: **SetMedVolume** *Volume*

Modes: Amiga/Blitz

Description:

SetMedVolume changes the overall volume that the Med Library plays the module, all the audio channels are affected. This is most useful for fading out music by slowly decreasing the volume from 64 to 0.

Function: **GetMedVolume**

Syntax: ***GetMedVolume*** *Channel#*

Modes: Amiga/Blitz

Description:

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

Function: **GetMedNote**

Syntax: **GetMedNote** *Channel#*

Modes: Amiga/Blitz

Description:

GetMedNote returns the current note playing from the specified channel. As with **GetMedVolume** this is useful for producing graphics effects synced to the music the Med Library is playing.

Function: **GetMedInstr**

Syntax: **GetMedInstr** *Channel*

Modes: Amiga/Blitz

Description:

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

Statement: **SetMedMask**

Syntax: **SetMedMask** *Channel Mask*

Modes: Amiga/Blitz

Description:

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

Serial Port Commands

The following are a set of commands to drive both the single RS232 serial port on an Amiga as well as supporting multiserial port cards such as the A2232 card. The unit# in the following commands should be set to 0 for the standard RS232 port, unit 1 refers to the default serial port set by the advanced serial preferences program and unit 2 on refer to any extra serial ports available.

Function: **OpenSerial**

Syntax: **OpenSerial** *unit#,baud,io_serflags*

Modes: Amiga

Description:

OpenSerial is used to configure a Serial Port for use. As with **OpenFile, OpenSerial** is a function and returns zero if it fails. If it succeeds advanced users may note the return result is the location of the IOExtSer structure. The baud rate should be in the range of 110-292,000. The io_serflags parameter includes the following flags:

```
bit7: #serf_xdisabled=128    ;disable xon/xoff
bit6: #serf_eofmode=64       ;enable eof checking
bit5: #serf_shared=32        ;set if you don't need exclusive use of port
bit4: #serf_rad_boogie=16    ;high speed mode
bit3: #serf_queuedbrk=8      ;if set a break command waits for buffer empty
bit2: #serf_7wire=4          ;if set use 7 wire RS232
bit1: #serf_parity_odd=2     ;select odd parity (even if not set)
bit0: #serf_parity_on=1      ;enable parity checking
```

Statement: **WriteSerial**

Syntax: **WriteSerial** *unit#,byte*

Modes: Amiga

Description:

WriteSerial sends one byte to the serial port. Unit# defines which serial port is used. If you are sending characters use the **Asc()** function to convert the character to a byte e.g. **WriteSerial** 0,**asc**("b").

Statement **WriteSerialString**

Syntax: **WriteSerialString** *unit#,string*

Modes: Amiga

Description:

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

Function: **ReadSerial**

Syntax: **ReadSerial** *(unit#) returns -1 if nothing waiting*

Modes: Amiga

Description:

ReadSerial returns the next byte waiting in the serial port's read buffer. If the buffer is empty it returns a -1. It is best to use a word type (var.w=ReadSerial(0)) as a byte will not be able to differentiate between -1 and 255.

Function: **ReadSerialString**

Syntax: **ReadSerialString** *(unit#) returns null if nothing waiting*

Modes: Amiga

Description:

ReadSerialString puts the serial port's read buffer into a string, if the buffer is empty the function will return a null string (length=0).

Statement: **CloseSerial**

Syntax: **CloseSerial** *unit#*

Modes: Amiga

Description:

The **CloseSerial** command will close the port, enabling other programs to use it. Note: Blitz will automatically close all ports that are opened when a program ends.

Statement SetSerialBuffer

Syntax: **SetSerialBuffer** *unit#,bufferlength*

Modes: Amiga

Description:

SetSerialBuffer changes the size of the ports read buffer. This may be useful if your program is not always handling serial port data or is receiving and processing large chunks of data. The smallest size for the internal serial port (unit#0) is 64 bytes. The bufferlength variable is in bytes.

Statement: SetSerialLens

Syntax: **SetSerialLens** *unit#,readlen,writelen,stopbits*

Modes: Amiga

Description:

SetSerialLens allows you to change the size of characters read and written by the serial device. Generally readlen=writelen and should be set to either 7 or 8, stopbits should be set to 1 or 2. Default values are 8,8,1.

Statement: SetSerialParams

Syntax: **SetSerialParams** *unit#*

Modes: Amiga

Description:

For advanced users, SetSerialParams tells the serial port when parameters are changed. This would only be necesary if they were changed by poking offsets from IOExtSer which is returned by the OpenSerial command.

Function: SerialEvent

Syntax: **SerialEvent** *(unit#)*

Modes: Amiga

Description:

SerialEvent is used when your program is handling events from more than 1 source, Windows, ARexx etc.

This command is currently not implemented

AREXX COMMANDS

Function: **CreateMsgPort()**

SYNTAX: PortAddress.l = **CreateMsgPort**(*"Name"*)

MODES:AMIGA

DESCRIPTION:

CreateMsgPort is a general Function and not specific to ARexx.

CreateMsgPort opens an intuition PUBLIC message port of the name supplied as the only argument. If all is well the address of the port created will be returned to you as a LONGWORD so the variable that you assign it to should be of type long.

If you do not supply a name then a private MsgPort will be opened for you.

. Port.l=**CreateMsgPort**("PortName")

It is important that you check you actually succeeded in opening a port in your program. The following code or something similar will suffice.

```
Port.l=CreateMsgPort("Name")
IF Port=0 THEN Error_Routine{}
```

The name you give your port will be the name that Arexx looks for as the HOST address,(and is case sensitive) so take this into consideration when you open your port. NOTE IT MUST BE A UNIQUE NAME AND SHOULD NOT INCLUDE SPACES.

DeleteMsgPort() is used to remove the port later but this is not entirely necessary as Blitz2 will clean up for you on exit if need be.

Statement: **DeleteMsgPort()**

STATEMENT: **DeleteMsgPort**

SYNTAX: DeleteMsgPort Port

MODES:AMIGA

DESCRIPTION:

DeleteMsgPort deletes a MessagePort previously allocated with **CreateMsgPort()**. The only argument taken by **DeleteMsgPort** is the address returned by **CreateMsgPort()**. If the Port was a public port then it will be removed from the public port list.

```
Port.l=CreateMsgPort("Name")
IF Port=0 Then End
DeleteMsgPort Port
```

Error checking is not critical as if this fails we have SERIOUS PROBLEMS.

YOU MUST WAIT FOR ALL MESSAGES FROM AREXX TO BE RECEIVED BEFORE YOU DELETE THE MSGPORT. IF YOU NEGLECT TO DELETE A MSGPORT BLITZ2 WILL DO IT FOR YOU AUTOMATICALLY ON PROGRAM EXIT.

Function: **CreateRexxMsg()**

SYNTAX: msg.l=**CreateRexxMsg**(*ReplyPort,"exten","HOST"*)

MODES:AMIGA

DESCRIPTION:

CreateRexxMsg() allocates a special Message structure used to communicate with Arexx. If all is successful it returns the LONGWORD address of this rexxmsg structure.

The arguments are *ReplyPort* which is the long address returned by **CreateMsgPort()**. This is the Port that ARexx will reply to after it has finished with the message.

EXTEN which is the exten name used by any ARexx script you are wishing to run. i.e. if you are attempting to run the ARexx script test.rexx you would use an *EXTEN* of "rexx".

HOST is the name string of the HOST port. Your program is usually the HOST and so this equates to the name you gave your port in **CreateMsgPort()**. REMEMBER IT IS CASE SENSITIVE.

As we are allocating resources error checking is important and can be achieved with the following code:

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

Statement: **DeleteRexxMsg**

SYNTAX: **DeleteRexxMsg** *rexxmsg*

MODES:AMIGA

DESCRIPTION:

DeleteRexxMsg simply deletes a RexxMsg Structure previously allocated by **CreateRexxMsg().** It takes a single argument which is the long address of a *RexxMsg* structure such as returned by **CreateRexxMsg()**.

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

Again if you neglect to delete the RexxMsg structure Blitz2 will do this for you on exit of the program.

Statement: **ClearRexxMsg**

SYNTAX: **ClearRexxMsg**1k

MODES:AMIGA

DESCRIPTION:

ClearRexxMsg is used to delete and clear an ArgString from one ormore of the Argument slots in a RexxMsg Structure. This is most useful for the more advanced programmer wishing to take

advantage of the Arexx #RXFUNC abilities.

The arguments are a LONGWORD address of a RexxMsg structure. **ClearRexxMsg** will always work from slot number 1 forward to 16.

```
Port.l=CreateMsgPort("TestPort")
If Port = NULL Then End
msg.l=CreateRexxMsg(Port,"vc","TestPort")
If msg=NULL Then End
SendRexxCommand msg,"open",#RXCOMM | #RXFF_RESULT
wait:WHILE GetMsg_(Port) <> msg:Wend        ;Wait for reply to come
ClearRexxMsg msg          .          ;Delete the Command string we sent
```

NOTE: ClearRexxMsg() is called automatically by RexxEvent() so the need to call this yourself is removed unless you have not sent the RexxMsg to Arexx.

Statement: **FillRexxMsg()**

SYNTAX: **FillRexxMsg** *rexxmsg,&FillStruct*

MODES:AMIGA

DESCRIPTION:

FillRexxMsg allows you to fill all 16 ARGSlots if necessary with either ArgStrings or numerical values depending on your requirement.

FillRexxMsg will only be used by those programmers wishing to do more advanced things with Arexx, including adding libraries to the ARexx library list, adding Hosts,Value Tokens etc. It is also needed to access Arexx using the #RXFUNC flag.

The arguments are a LONG Pointer to a *rexxmsg*.

The LONG address of a FillStruct NEWTYPE structure. This structure is defined in the Arexx.res and has the following form.

```
NEWTYPE.FillStruct
  Flags.w        ;Flag block
  Args0.l        ; argument block (ARG0-ARG15)
  Args1.l        ; argument block (ARG0-ARG15)
  Args2.l        ; argument block (ARG0-ARG15)
  Args3.l        ; argument block (ARG0-ARG15)
  Args4.l        ; argument block (ARG0-ARG15)
  Args5.l        ; argument block (ARG0-ARG15)
  Args6.l        ; argument block (ARG0-ARG15)
  Args7.l        ; argument block (ARG0-ARG15)
  Args8.l        ; argument block (ARG0-ARG15)
  Args9.l        ; argument block (ARG0-ARG15)
  Args10.l       ; argument block (ARG0-ARG15)
  Args11.l       ; argument block (ARG0-ARG15)
  Args12.l       ; argument block (ARG0-ARG15)
  Args13.l       ; argument block (ARG0-ARG15)
  Args14.l       ; argument block (ARG0-ARG15)
  Args15.l       ; argument block (ARG0-ARG15)
  EndMark.l      ;End of the FillStruct
End NEWTYPE
```

The Args?.l are the 16 slots that can possibly be filled ready for converting into the RexxMsg structure. The Flags.w is a WORD value representing the type of LONG word you are supplying

for each ARGSLOT (Arg?.l).

Each bit in the Flags WORD is representative of a single Args?.l, where a set bit represents a numerical value to be passed and a clear bit represents a string argument to be converted into a ArgString before installing in the RexxMsg. The Flags Value is easiest to supply as a binary number to make the bits visible and
would look like this.

%0000000000000000 ;This represents that all Arguments are Strings.

%0110000000000000 ;This represent the second and third as being integers.

FillRexxMsg expects to find the address of any strings in the Args?.l slots so it is important to remember when filling a FillStruct that you must pass the string address and not the name of the string. This is acomplished using the '&' address of operand.

So to use FillRexxMsg we must do the following things in our program:

1. Allocate a FillStruct
2. Set the flags in the FillStruct\Flags.w
3. Fill the FillStruct with either integer values or the
   addresses of our string arguments.
4. Call FillRexxMsg with the LONG address of our rexxmsg and the
   LONG address of our FillStruct.

To accomplish this takes the following code:

```
;Allocate our FillStruct (called F)

DEFTYPE.FillStruct F

;assign some string arguments

T$="open":T1$="0123456789"

;Fill in our FillStruct with flags and (&) addresses of our strings

F\Flags= %0010000000000000,&T$,&T1$,4

;Third argument here is an integer (4).

Port.l=CreateMsgPort("host")
msg.l=CreateRexxMsg(Port,"vc","host")

FillRexxMsg msg,&F

;<-3 args see #RXFUNC

SendRexxCommand msg,"",#RXFUNC I #RXFF_RESULT I 3
```

Function: **CreateArgString()**

SYNTAX: ArgString.l=**CreateArgString**(*"this is a string"*)

MODES:AMIGA

DESCRIPTION:

CreateArgString() builds an ARexx compatible ArgString structure around the provided string.

All strings sent to, or received from Arexx are in the form of ArgStrings. See the TYPE RexxARG.

If all is well the return will be a LONG address of the ArgString structure. The pointer will actually point to the NULL terminated String with the remainder of the structure available at negative offsets.

```
arg.l=CreateArgString("this is a string")
 IF arg=0 THEN Error_Routine{}:ENDIF
DeleteArgString arg
```

NOTE: An ArgString maybe used as a normal BB2 string variable by simple conversion using **PEEK$**

i.e. msg$=**PEEK$**(arg) or perhaps **NPRINT PEEK$**(arg)

NOTE: Most of the BB2 Arexx Functions call this themselves and there will be only limited need for you to access this function.

Statement: **DeleteArgString**

SYNTAX:**DeleteArgString** *ArgString*

MODES:AMIGA

DESCRIPTION:

DeleteArgString is designed to Delete ArgStrings allocated by either Blitz2 or ARexx in a system friendly way. It takes only one argument the LONGWORD address of an ArgString as returned by **CreateArgString().**

```
arg.l=CreateArgString("this is a string")
 IF arg=0 THEN Error_Routine{}:ENDIF
DeleteArgString arg
```

NOTE: This function is also called automatically by most of the BB2 Arexx Functions that need it so you should only need to call this on rare occations.

Statement: **SendRexxCommand**

SYNTAX: **SendRexxCommand** *rexxmsg,"commandstring",#RXCOMM| #RXFF_RESULT*

MODES:AMIGA

DESCRIPTION:

SendRexxCommand is designed to fill and send a RexxMsg structure to ARexx inorder to get ARexx to do something on your behalf.

The arguments are as follows;

rexxmsg is the LONGWORD address of a RexxMsg structure as returned by CreateRexxMsg().

commandstring is the command string you wish to send to ARexx. This is a string as in "this is a string" and will vary depending on what you wish to do with ARexx. Normally this will be the name of an ARexx script file you wish to execute. ARexx will then look for the script by the name as well as the name with the exten added.(this is the exten you used when you created

the RexxMsg structure using CreateRexxMsg()). This could also be a string file. That is a complete ARexx script in a single line.

ActionCodes are the flag values you use to tell ARexx what you want it to do with the commandstring you have supplied. The possible flags are as follows;

COMMAND (ACTION) CODES

The command codes that are currently implemented in the resident process are described below. Commands are listed by their mnemonic codes,followed by the valid modifier flags. The final code value is always the logical OR of the code value and all of the modifier flags selected. The command code is installed in the rm_Action field of the message packet.

USAGE: RXADDCON

This code specifies an entry to be added to the Clip List. Parameter slot ARGO points to the name string,slot ARG1 points to the value string,and slot ARG2 contains the length of the value string.

The name and value arguments do not need to be argstrings,but can be just pointers to storage areas. The name should be a null-terminated string,but the value can contain arbitrary data including nulls.

USAGE: RXADDFH

This action code specifies a function host to be added to the Library List. Parameter slot ARGO points to the (null-terminated) host name string,and slot ARG1 holds the search priority for the node. The search priority should be an integer between 100 and -100 inclusive;the remaining priority ranges are reserved for future extensions. If a node already exists with the same name,the packet is returned with a warning level error code.

Note that no test is made at this time as to whether the host port exists.

USAGE:RXADDLIB

This code specifies an entry to be added to the Library List. Parameter slot ARGO points to a null-terminated name string referring either to a function library or a function host. Slot ARG1 is the priority for the node and should be an integer between 100 and -100 inclusive;the remaining priority ranges are reserved for future extensions. Slot ARG2 contains the entry Point offset and slot ARG3 is the library version number. If a node already exists with the same name,the packet is returned with a warning level error code. Otherwise,a new entry is added and the library or host becomes available to ARexx programs. Note that no test is made at this time as to whether the library exists and can be opened.

USAGE:RXCOMM [RXFF_TOKEN] [RXFF_STRING] [RXFF_RESULT] [RXFF_NOIO]

Specifies a command-mode invocation of an ARexx program. Parameter slot ARGO must contain an argstring Pointer to the command string. The RXFB_TOKEN flag specifies that the command line is to be tokenized before being passed to the invoked program. The RXFB_STRING flag bit indicates that the command string is a "string file." Command invocations do not normally return result strings,but the RXFB_RESULT flag can be set if the caller is prepared to handle the cleanup associated with a returned string. The RXFB_NOIO modifier suppresses the inheritance of the host's input and output streams.

USAGE:RXFUNC [RXFF_RESULT] [RXFF_STRING] [RXFF_NOIO] argcount

This command code specifies a function invoction. Parameter slot ARGO contains a pointer to the function name string,and slots ARG1 through ARG15 point to the argument strings,all of which must be passed as argstrings. The lower byte of the command code is the argument count;this count excludes the function name string itself. Function calls normally set the RXFB_RESULT flag,but this is not mandatory. The RXFB_STRING modifier indicates that the function name string

is actually a "string file". The RXFB_NOIO modifier suppresses the inheritance of the host's input and output streams.

USAGE:RXREMCON

This code requests that an entry be removed from the Clip List. Parameter slot ARGO points to the null-terminated name to be removed. The Clip List is searched for a node matching the supplied name,and if a match is found the list node is removed and recycled. If no match is found the packet is returned with a warning error code.

USAGE:RXREMLIB

This command removes a Library List entry. Parameter slot ARGO points to the null terminated string specifying the library to be removed. The Library List is searched for a node matching the library name,and if a match is found the node is removed and released. If no match is found the packet is returned with a warning error code. The libary node will not be removed if the library is currently being used by an ARexx program.

USAGE:RXTCCLS

This code requests that the global tracing console be closed. The console window will be closed immediately unless one or more ARexx programs are waiting for input from the console. In this event,the window will be closed as soon as the active programs are no longer using it.

USAGE:RXTCOPN

This command requests that the global tracing console be opened. Once the console is open,all active ARexx programs will divert their tracing output to the console. Tracing input(for interactive debugging)will also be diverted to the new console. Only one console can be opened;subsequent RXTCOPN requests will be returned with a warning error message.

MODIFIER FLAGS

Command codes may include modifier flags to select various processing options. Modifier flags are specific to certain commands,and are ignored otherwise.

RXFF_NOIO.

This modifier is used with the RXCOMM and RXFUNC command codes to suppress the automatic inheritance of the host's input and output streams.

RXFF_NONRET.

Specifies that the message packet is to be recycled by the resident process rather than being returned to the sender. This implies tht the sender doesn't care about whether the requested action succeeded,since the returned packet provides the only means of acknowledgement. (RXFF_NONRET MUST NOT BE USED AT ANY TIME)

RXFF_RESULT.

This modifer is valid with the RXCOMM and RXFUNC commands,and requests that the called program return a result string. If the program EXITs(or RETURNs)with an expression,the expression result is returned to the caller as an argstring. This ArgString then becomes the callers responsibility to release. This is automatically accomplished by using GetResultString(). It is therefore imperitive that if you use RXFF_RESULT then you must use GetResultString() when the message packet is returned to you or you will incure a memory loss equal to the size of the ArgString Structure.

RXFF_STRING.

This modifer is valid with the RXCOMM and RXFUNC command codes. It indicates that the

command or function argument(in slot ARGO)is a "string file" rather than a file name.

RXFF_TOKEN.

This flag is used with the RXCOMM code to request that the command string be completely tokenized before being passed to the invoked program. Programs invoked as commands normally have only a single argument string. The tokenization process uses "white space" to separate the tokens,except within quoted strings. Quoted strings can use either single or double quotes,and the end of the command string(a null character) is considered as an implicit closing quote.

EXAMPLES:

```
Port.l=OpenRexxPort("TestPort")
  If Port = NULL End:EndIf
msg.l=CreateRexxMsg(Port,"vc","TestPort")
  If msg=NULL End:EndIf
SendRexxCommand msg,"open",#RXCOMM | #RXFF_RESULT
```

Statement: **ReplyRexxMsg**

SYNTAX: **ReplyRexxMsg** *rexxmsg,Result1,Result2,"ResultString"*

MODES:AMIGA

DESCRIPTION:

When ARexx sends you a RexxMsg (Other than a reply to yours i.e. sending yours back to you with results) you must repl to the message before ARexx will continue or free that memory associated with that RexxMsg. ReplyRexxMsg accomplishes this for you. ReplyRexxMsg also will only reply to message that requires a reply so you do not have to include message checking routines in your source simply call ReplyRexxMsg on every message you receive wether it is a command or not.

The arguments are;

rexxmsg is the LONGWORD address of the RexxMsg Arexx sent you as returned by GetMsg_(Port).

Result1 is 0 or a severity value if there was an error.

Result2 is 0 or an Arexx error number if there was an error processing the command that was contained in the message.

ResultString is the result string to be sent back to Arexx. This will only be sent if Arexx requested one and Result1 and 2 are 0.

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

Function: **GetRexxResult()**

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

MODES:AMIGA

DESCRIPTION:

GetRexxResult extracts either of the two result numbers from the RexxMsg structure. Care must be taken with this Function to ascertain wether you are dealing with error codes or a ResultString address. Basically if result 1 is zero then result 2 will either be zero or contain a ArgString pointer to the ResultString. This should then be obtained using **GetResultString()**.

The arguments to **GetRexxResult** are;

rexxmsg is the LONGWORD address of a RexxMsg structure returned from ARexx.

ResultNum is either 1 or 2 depending on wether you wish to check result 1 or result 2.

```
;print the severity code if there was an error

NPrint GetRexxResult(msg,1)

;check for ResultString and get it if there is one

IF GetRexxResult(msg,1)=0
 IF GetRexxResult(msg,2) THEN GetResultString(msg)
ENDIF
```

Function: **GetRexxCommand()**

SYNTAX: *String$*=**GetRexxCommand**(*msg, 1*)

MODES:AMIGA

DESCRIPTION:

GetRexxCommand allows you access to all 16 ArgString slots in the given RexxMsg. Slot 1 contains the command string sent by ARexx in a command message so this allows you to extract the Command.

Arguments are:

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

ARGNum is an integer from 1 to 16 specifying the ArgString Slot you wish to get an ArgString from.

BEWARE YOU MUST KNOW THAT THERE IS AN ARGSTRING THERE.

Function: **GetResultString()**

SYNTAX: String$=GetResultString*(rexxmsg)*

MODES:AMIGA

DESCRIPTION:

GetResultString allows you to extract the result string returned to you by ARexx after it has completed the action you requested. ARexx will only send back a result string if you asked for one (using the ActionCodes) and the requested action was successful.

```
;check for ResultString and get it if there is one

IF GetRexxResult(msg,1)=0
  IF GetRexxResult(msg,2) THEN GetResultString(msg)
ENDIF
```

NOTE: Do not attempt to DeleteArgString the result string returned by this function as the return is a string and not an ArgString pointer. BB2 will automatically delete this argstring for you.

Statement: **Wait**

SYNTAX: **Wait**

MODES:AMIGA

DESCRIPTION:

Wait halts all program execution until an event occurs that the program is interested in. Any intuition event such as clicking on a gadget in a window will start program execution again.

A message arriving at a MsgPort will also start program execution again. So you may use **Wait** to wait for input from any source including messages from ARexx to your program.

Wait should always be paired with **EVENT** if you need to consider intuition events in your event handler loop.

```
Repeat
  Wait:rmsg.l=REXXEVENT(Port):ev.l=EVENT
  IF IsRexxMsg(Rmsg) Process_Rexx_Messages{}:ENDIF
  ;
  ;
  ;Rest of normal intuition event loop statements case etc
  ;
Until ev =$200
```

Function: **RexxEvent()**

SYNTAX: Rmsg.l=**RexxEvent**(*Port*)

MODES:AMIGA

DESCRIPTION:

RexxEvent is our Arexx Equivalent of **EVENT()**. It's purpose is to check the given Port to see if there is a message waiting there for us.

It should be called after a **WAIT** and will either return a NULL to us if there was no message or the LONG address of a RexxMsg Structure if there was a message waiting.

Multiple Arexx MsgPorts can be handled using separate calls to RexxEvent():

```
Wait:Rmsg1.l=RexxEvent(Port1):Rmsg2.l=RexxEvent(Port2):etc
```

RexxEvent also takes care of automatically clearing the rexxmsg if it is our message being returned to us.

The argument is the LONG address of a MsgPort as returned by CreateMsgPort().

EXAMPLES:

```
Repeat
 Wait:Rmsg.l=REXXEVENT(Port):ev.l=EVENT
 IF IsRexxMsg(Rmsg) Process_Rexx_Messages{}:ENDIF
 ;
 ;
 ;
 Rest of normal intuition event loop statements case etc
Until ev =$200
```

SEE ALSO: Wait(),CreateMsgPort()

Function: **IsRexxMsg()**

SYNTAX: **IsRexxMsg***(rexxmsg)*

MODES:AMIGA

DESCRIPTION:

IsRexxMsg tests the argument (a LONGWORD pointer hopefully to a message packet) to see if it is a RexxMsg Packet. If it is TRUE is returned (1) or FALSE if it is not (0).

```
repeat
 Wait:Rmsg.l=REXXEVENT:ev.l=EVENT
 IF IsRexxMsg(Rmsg) Process_Rexx_Messages{}:ENDIF
 ;
 ;
 ;
 Rest of normal intuition event loop statements case etc
until ev =$200
```

As the test is non destructive and extensive passing a NULL value or a LONGWORD that does not point to a Message structure (Intuition or Arexx) will safely return as FALSE.

SEE ALSO: CreateRexxMsg(),GetMsg_()

Function: **RexxError()**

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

MODES:AMIGA

DESCRIPTION:

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

```
NPRINT RexxError(5)
```

SEE ALSO: GetRexxResult()

SIMON'S 3D LIBRARY

The following are pre-release docs for the 3D library featured on the cover disk.

Statement: **InitDisplay3D**

Syntax: **InitDisplay3D** *Display3D#,BitMap0#,BitMap1#,flags,xview%,yview%,[,x,y,w,h]*

Modes: Amiga/Blitz

Description:

InitDisplay3D initialises a display system for the 3d library. A 3d display can either be the size of the bitmap specified, or by adding the optional parameters a window in the bitmap.

Windows must be located on a 16 bit boundary and be a multiple of 16 pixels wide.

The two BitMaps supplied must be the same size, if double buffering is not required *BitMap1#* can be the same as *BitMap#0.*

The *flags* parameter contains the following bits:

#STEREO=1 Configure a stereoscopic display, bitmaps supplied must be twice the height of the display, the 3d library then renders the left eye image in the top half and the right eye image in the lower half when DrawScene3D is called

#LUT=2 Add look up table to the Display3D object, this takes 32K of memory, speeds up the projection algorithm used by Drawscene3D, however rendering is not as accurate as a display with no lut.

The field of vision is defined by *xview%* and *yview%*. 50% will display 90 degrees in front of the viewer, 40% will display 72 degrees (recomended).

Statement: **FreeDisplay3D**

Syntax: **FreeDisplay3D** *Display3D#*

Modes: Amiga/Blitz

Description:

FreeDisplay3D will free the memory allocated by the *Display3D#.* 3D displays are automatically freed when the program ends.

Statement: **Horizon3D**

Syntax: **Horizon3D** *GroundColor,SkyColor*

Modes: Amiga/Blitz

Description:

The **Horizon3D** command is used to configure a horizon for use by the 3D library and define the colors from the current palette that are used for ground and sky. If GroundColor=0 no horizon is drawn by DrawScene3D. GroundColor must be an odd number and SkyColor must be even.

Statement: **DrawScene3D**

Syntax: **DrawScene3D** *Display3D#,Buffer#,ShapeList() ;camera=current item*

Modes: Amiga/Blitz

Description:

DrawScene3D is used to render the 3D scene onto the specified 3D display. *Buffer#* can be either 0 or 1 and relates directly to the bitmaps that were specified when the Display3D was initialised.

The *ShapeList()* should be a list of .part3D's. The current item in the list will be used as the viewer and it's position and orientation are used to calculate the other objects position in the scene. The current item should be thought of as the camera.

Statement: **SetStereo3D**

Syntax: **SetStereo3D** *Display3D#,focal distance,separation*

Modes: Amiga/Blitz

Description:

If the Display3D is configured with #STEREO set in flags the SetStereo3D command is used to adjust the focal distance and separation amount. The focal length is the distance from the viewer an objects left and right eye image will be in the same place, the separation variable defines how much the two images will diverge as the object becomes closer or further away.

Statement: **MoveShapes3d**

Syntax: **MoveShapes3d** *ShapeList()*

Modes: Amiga/Blitz

Description:

The **MoveShapes3D** command will process all the shapes in the *Shapelist()* provided, the anim[n] varaiables define how the objects are moved as well as the speed and acceleration variables. All children of the shapes in the list are moved as well.

THE 3D SHAPE STRUCTURE

The .part3D type is used to hold information about each object in the 3D universe. A list array of shapes is created, and the commands DrawScene3D and MoveShapes3D process the entire list when they are called, this is much faster than calling the routines for each object.

The .part3D type:

```
NEWTYPE .xyz:x.q:y:z:End NEWTYPE

NEWTYPE .matrix:m.w[9]:End NEWTYPE

NEWTYPE .part3d
  *sister.part3d
  *child.part3d
  *parent.part3d
  *frame.w
  rota.xyz:rotv:rot
  posa.xyz:posv:pos
  id.matrix
  veepos.w[3]
  view.matrix
  animval.w[16]
End NEWTYPE
```

The first 3 pointers are used for linking shapes together. This should be tackled only when the programmer is familiar with all the other features of Blitz3D. Briefly, if a shape has a rotating radar on its roof, the radar would be defined outside the main shape list, *child would point to the separate radar shape, and its *parent would point back to the shapes location. The position and orientation of the radar is then calculated relative to the parent shape. More details will be published in the next issue.

The *frame field should point to the object's shape (see next section).

The rota, rotv and rot triplets represent the objects rotation (orientation), rotv represents the rotational velocity and rota it's acceleration.

The posa, posv and pos triplets represent the objects position posv represents its velocity and posa its acceleration.

The id.matrix is used internally and holds the transformation matrix for the shape, this is calculated from it's rot value.

The veepos and view.matrix are used internally and hold the shapes relative position & orientation to the viewer.

The animvals contain information for the MoveShapes3D routine. If animval[0] is negative the value is calculated by 2^parameter this means the algorithm uses shifts not multiplies which is much faster.

animval[0]	0=stationary object, positive=constants, negative=shifts
animval[1]	thrust, fraction of z heading added to shapes acceleration each move
animval[2]	drag, fraction of speed subtracted from objects speed each move
animval[3]	rotddrag, fraction of rotational speed subtracted each move

The drag variables mean that you can limit the maximum amount of velocity an object

reaches when it is constantly accelerating.

3D FRAME STRUCTURE

The *frame field in the shape must point to a valid frame, this is a collection of verticies and polygon descriptions that describe what the shape looks like.

Depending on the size of the shape you will want to adjust inveiw so it disappears from sight when it is so far from the camera, and tooclose so the 3D library does not try and draw it when some verticies may be behind you. Sorry, no z axis clipping yet.

Use the shift version for fast verticy generation, normal for the values that are not exponents of 2. The origin is v0 code $00, v1 is calulated at $10, v2 at $20 etc.

The extrude lists have not been extensively tested, more explanations next issue, however they will save lots of time for verticy generation.

The polygons are described by number of verticies, flags then verticy codes: $00=origin, $10 first verticy generated in list...

The color is defined in c0-c3, these 4 bytes directly relate to the colors used in a 2x2 dithering matrix. Keep c0-c3 the same for a solid color, a mix of two colors should be listed as co0,co1,co1,co0 and a mix of 4 colors listed in any order. The variables f0 and f1 will be for planepick, transparency, shading surface detail which I will endeavour to add for next issue along with spheres.

The whole frame format is specified as follows:

```
nxframe.l   ;reserved for animating frames
pvframe.l   ;reserved for animating frames
inview.w    ;distance from camera when out of view
tooclose.w  ;distance from camera when too close
type.w      ;0=polygons ....

;verticy list in shift style -3=>-8 -2=>-4 -1=>-1 0=>0 1=>1 2=>4 3=>8
numvshifts.w [,x.w,y.w,z.w]...

;verticy list in normal style
numvnormal.w [x.w,y.w,z.w]...

;single extrusion of point v0 in steps of v1 num times
numvrepeats.w [v0.w,v1.w,num.w]...

;extrude n1 set of points located at v0 in direction v1 n2 times
numvextrudes [v0.w,v1.w,n1.w,n2.w]..

;list polygons in clockwise order
[vnum.w,flags,v.w.........v.w,c0.b,c1.b,c2.b,c3.b,f0.w,f1.w]..

;end with two zeros.
dc 0,0
```

Apologies for not covering the 3D library in more depth, the demos on the cover disk should illustrate most of the concepts, the rest will just have to wait for the next issue of Blitz User, stay tuned!

LIBRARY DEVELOPMENT

The following is an extended discussion of topics discussed in chapter 5 of the User Guide. This is for users with experience in machine code wanting to take advantage of the powerful Blitz 2 library system. All the constants and macros used are defined in the libmacs file found in the blitzlibs: volume.

A lot of this reference material is complex, we will endeavour to publish examples of most of this material in the next issue of BlitzUser. We are confident that the way we have engineered the Blitz 2 library system will mean Blitz will continue to grow in leaps and bounds.

THE LIBRARY HEADER

```
#mylib=50
;
!libheader {#mylib,init,1,finit,0}
;
!dumtoke{"MyModuleName","",_toke}
;
!astatement !args {#word,#string}
!libs {#medlib,$1380,#doslib,#la6}
!subs {_loadmyobject+1,0,0}
!name {"LoadMyObject","MyObject#,FileName$"}
;
; more statements and functions go here
;
load:!nullsub{0,0,0}
save:!nullsub{0,0,0}
use:!nullsub{0,0,0}
free:!nullsub{_freemyobject,0,0}
;
init:!nullsub{_initmylib,0,0}
finit:!nullsub{_closemylib,0,0}
;
!libfin{_toke,_load,_save,_use,_free,8,5}
```

INIT/FINIT

When Blitz2 first runs a program it calls all the init routines of the libraries required by the program. Any code to initialise tables and allocate memory for the libraries' routines would be pointed to by the init nullsub in this instance a routine called _initmylib.

The 1 after init in libheader means our init routine returns a long word for use by other libraries (must be lower priority). This is useful for sharing large tables or a common data structure between two related libraries.

The finit routine is called when a program ends, it is useful for any clean up procedures,

for example the med library uses a finit routine to make sure all audio channels are silent, it could also be used to make sure specific opened files are closed etc.

ERROR CHECKING

When error checking is enabled Blitz 2 calls any specified error routines before calling commands. The last parameter of libheader should point to the start of these routines. When error checking is not enabled Blitz 2 will not load this section of the library saving memory.

If any commands require an optional error checking routine that is enabled/disabled from the compiler options the location of the error checking routine should be passed in the second parameter of the !subs macro of that command. The error routine should be located at the end of the library after the errorroutines pointer in !libheader.

An error routine usually comprises of code that checks parameters are in range, if it fails the location of a message is placed in register d0 followed by a Trap#0 instruction which will bring up the familiar Blitz 2 error requester, if parameter checking succeeds the error routine should return making sure not to corrupt any vital registers.

LIBRARY OBJECTS

The optional parameters of the !libfin macro are used to define a libraries' own object type. Examples of Blitz objects are listed in Appendix 1 of the reference manual.

The first parameter of !libfin points to the !dumtoke contining the objects name. This dumtoke should be positioned directly after !libinit as in the above example. The Load and Save routines are not implemented and should point to nullsub macros as listed.

The used nullsub can point to a routine which will be called when the Use MyObjectName command is used. An example of this is Use Palette 3, not only does Palette Object number 3 become the currently used object but is also displayed on the current Slice or Screen.

The Free nullsub can be used to de-allocate memory that the object used.

The last two parameters of !libfin are the default number of objects that will be allocated (can be changed by the user in the compiler options requester) and the size of each object. The size is computed by 2^sizeparameter so the 5 in the above example represents 32 bytes allocated for each object.

OBJECT PARAMETERS

The Blitz 2 library system has a wide range of features in the parameters (arguments) department.

The !args macro can contain up to 15 argument definitions. To specify an argument use the constants #byte, #word, #long, #quick and #float. Blitz 2 will convert the type the user passed to the correct type.

The parameters will be passed in registers d0-d5, extra parameters will be passed to your routine on the stack and can be recovered by move(.l)-(a2),register.

The following argument definitions can sometimes be used in combination, for instance the sort command uses #array+#arrayend+ #unknown so it can operate on any type of single dimension array of any type. The array base arrives in d0, an the routine pulls the array end location of the stack first and then the type.

#string specifies a string argument, the location is passed in the proper register and its length will be pushed on a stack, recovered by a move.l -(a2),register. All Blitz strings are null terminated.

#usesize must be the first parameter definition (if required) and will return the type of command in register d0. i.e. poke.w returns the constant #word in d0.

#unknown will pass the type of variable the user passed to the routine on the stack and can be recovered by move.w -(a2),register. The print statement uses this form to evaluate what type each of the paramters passed to it are.

#array will pass the location of the array/list base to your routine.

#arrayend will pass the location of the end of the array on the stack, used by the sort() command to calculate number of items in the array

#push will place the parameter on the stack and not in the relevant data register (useful for calling C routines).

#varptr returns the location of the variable passed (not its value).

Variable specification of parameters are possible with the !repargs macro. The format is similar to the standard !args macro however two parameters are inserted at the start of the code, first_repeat and size_repeat. Blitz 2 will treat the next argument definitions as usual until the first_repeat parameter is reached and then parameter definitions apply to extraneous parameters in groups of size_repeat.

The following examples should make things clearer...

```
myroutine a.w,b.q [,cn.w...]      => !repargs{2,1,#word,#quick,#word}
myroutine a.w [[,t1n.q,t2.w]...]  => !repargs{1,2,#word,#quick,#word}
```

The second example expects a word then any number of pairs of quick,word. Blitz 2 will pass your routine the number of parameters in register d7.

PROCESSING LISTS

The #array parameter definition also enables the base of a ListArray to be passed to your routines. The characters "llsT" ($6c497354) are located at -12(arraybase) if the array is a link list. The following code example demonstrates processing the contents of a linked list, presuming its base is passed in d0....

```
     MOVE.l d0,a0        ;a0=list base
     MOVE.l -28(a0),a0   ;a0 points to first item (-32= *current item)
nxt: TST.l (a0)          ;empty node?
     BEQ   finished      ;yes exit
     LEA   8(a0),a1      ;a1 points to item data
     BSR   process_item
     MOVE.l (a0),a0      ;go to next item in list (4(a0)= *previous item)
     BRA   nxitem
dun: RTS
```

THE LIBS MACRO

Often your routines will require a library base (value returned by that libraries init routine) or pointers to library objects (be it your own libraries objects or other libraries).

The following constants are for use in the !libs macro and define what information you require passed in which register. reg is restricted to d0-d7/a0-a3/a6. A library number should follow each constant be it your own libraries or another.

#l*reg*,libnum load the specific register with libnum's base

#lpush,libnum push the libnum's base onto the stack

#u*reg*,libnum loads *reg* with location of currently used library object

#b*reg*,libnum loads *reg* with location of base item (item 0)

#m*reg*,libnum returns the maximum number of objects set in compiler options

#i*reg*l #p*reg*,libnum loads the i*reg* with location of object(#p*reg*), #p*reg* being a parameter passed in args.

Examples:

!libs{#la1,#chipbaselib,#la6,#doslib}

Location $dff000 is passed in a1 ,_DosBASE in a6 (ready for dos calls)

!libs{#ia0l #pd0,#shapeslib,#ua1,#bitmaplib}

The location of the shape with number specified by the first argument is passed in a0, the currently used bitmap is passed in a1.a

BLITZ/AMIGA MODES

If a 1 is added to the first parameter of a !subs macro Blitz will only let the user use the command in Amiga mode as in the above example. If a 1 is added to the second parameter (error routine) then the command is BlitzMode only. Of course any 1 is stripped by Blitz when it evaluates the labels!

If the command has two routines one for Blitz mode and one for Amiga mode the following convention should be used at the routines labelled location:

```
routine:        dc.w   $a0001        ;call blitz 2 line 1010 emulator
                dc.l   blitz_routine
amiga_routine:  ;
                ;amiga version here
                ;
                rts
blitz_routine:  ;
                ;blitz version here
                ;
                rts
```

If your command uses the blitter the Amiga version of the command should call

Own_Blitter, bsr the BlitzMode version then DisOwn_Blitter. Before poking Blitter registers in your routine always do a BlitWait. The following code should be used:

```
myblit:  dc.w      $a001
         dc.l      _domyblit       ;in blitz mode, go straight there
         ALibJSR   $c203           ;own blitter
         bsr       _domyblit
         ALibJSR   $c204           ;disown blitter
         rts
domyblit:
```

INLINE CODE

PC Relative code can be inserted directly into the object code by the Blitz 2 compiler. Note the rts is not included in the size of the code. The code must be completely self contained with no absolute addressing.

```
commandlabel:   dc    $a000,'f-'s
's                    ;
                      ; code goes here (no absolute adressing!)
'f                    ;
                rts
```

CALLING OTHER LIBRARIES

The memory allocate and free routines have already been described in the Libraries chapter of the User Guide. The following is a list of useful System calls for use with the ALibJsr command (the BLibJsr command is for accessing the BlitzMode version of certain commands).

All of the following routines will only alter the registers supplied and preserve all registers not listed.

Command: **#globalalloc**=$c002 (memlib)
Syntax: memoryblock d0.l = **globalalloc** (bytesize d0.l, memtype d1.l)
Description: Standard memory allocate routine, automatically cleared by Blitz when program ends, see also globalfree.

Command: **#globalfree**=$c003 (memlib)
Syntax: **globalfree** memorylocation a1.l, bytesize d0.l
Description: Standard memory deallocate routine.

Command: **#newlocalmem**=$c004 (memlib)
Description: Creates a new local memory node, which will link subsequent calls to localalloc together, useful for recursive type operations, for advanced users only.

Command: **#freelocalmem**=$c005 (memlib)
Description: Frees all memory attatched to current local memory node.

Command: **#localalloc**=$c000 (memlib)
Syntax: memoryblock d0.l = localalloc (bytesize d0.l, memtype d1.l)
Description: Local memory allocate routine, for use after newlocalmem call.

Command: **#localfree**=$c001 (memlib)
Syntax: **localfree** (memorylocation a1.l, bytesize d0.l)

Description: Local memory deallocate routine.

Command: **#addanint**=$c100 (intlib)
Syntax: **addanint** level+ID+$8000 d0.w, code d1.l
Description:
Adds an interupt at the level specified, low 4 bits of d0 should contain interupt level, high bit must be set, other bits can be used for ID. Interupt code should be pointed to by register d1.
It is up to the programmer to determine whether ALibJSR of BLibJSR is appropriate by determining which mode their command is being called in. Amiga mode interupt code MUST preserve d2-d7/a2-a4 and end with moveq#0,d0, Blitz mode interupts should preserve registers d5-d7/a4.

Command: **#clranint**=$c101 (intlib)
Syntax: **clranint** level+ID+$8000 d0.w
Description:
Removes the interupt(s) with specified ID from the interupt list.
Reserved Interupt ID's are:
Level 3: $8003-BlitzKeys (strobe)
Level 5: $8005-FadeLib
$8015-Mouse
$8025-TrackerLib
$8035-BlitzKeys (repeat function)

Commands: **#goblitz**=$c200, #goamiga=$c201, #goqamiga=$c202 (switchlib)
Description: Changes operating mode for compiler and program.

Commands: **#ownblit**=$c203, #disownblit=$c204 (switchlib)
Description: Must be used before and after routines that use the blitter in Amiga mode.

Command: **#progend**=$c800 (exitslib)
Description: same as the Blitz command End.

Command: **#getffpbase**=$c900 (ffplib)
Description: returns library base of mathffp.library in a6.

Commands: **#quickmult**=$ca00, **#longmult**=$ca01 (lmulllib)
Syntax: d0.q=**quickmult**(d0.q,d1.q) and d0.l=**longmult**(d0.l,d1.l)
Description: Functions that multiply two quicks and two longs.

Commands: **#quickdiv**=$cb00, **#longdiv**=$cb01 (ldivlib)
Syntax: d0.q=**quickdiv**(d0.q/d1.q) and d0.l=**longdiv**(d0.l/d1.l)
Description: Functions that divide two quicks and two longs.

Command: **#allocstring**=$cf01 (maxslib)
Syntax: string d0.l=**allocstring**(location of text d0.l, length d1.l)
Description:
A null terminated copy of the string is created, a pointer to which is returned in D0. This is mainly used to create copies of string parameters for such things as screen or window titles.

Command: **#freestring**=$cf02 (maxslib)
Description: frees up the string pointed to the register d0.

Commands: **#quicktofloat**=$d300, **#floattoquick**=$d301 (floatquicklib)
Syntax: d0.f=**quicktofloat**(d0.q) d0.q=**floattoquick**(d0.f)
Description: Functions to converts between quick and long types.