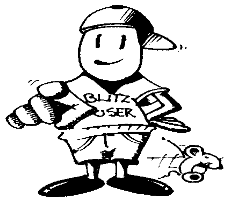
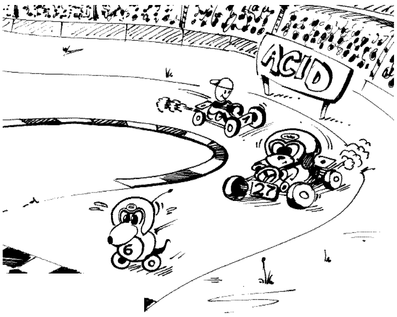
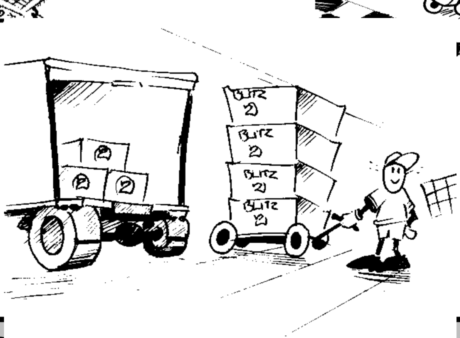
INFORMATION AND SUPPORT FOR BLITZ BASIC 2 USERS WORLD WIDE

BLITZ USER

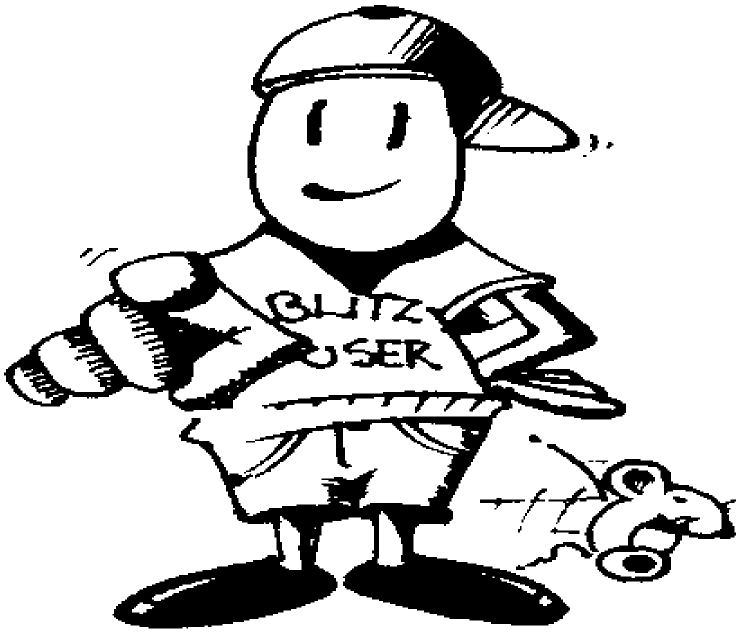


ISSUES 1 -5





A compilation of Blitz User Magazines 1-5
featuring 100+ new commands and more...



BLITZ MAN WANTS YOU!

SUBSCRIBE NOW

UPGRADES, ARTICLES, TUTORIALS, EXTRA COMMANDS AND MORE REGULARLY DELIVERED TO YOUR PLACE OF PROGRAMMING

SUBSCRIBE NOW

CONTENTS

Issues

1-5

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:

ACIDSOFTWARE
10 StKevins Arcade
Karangahape Rd
Auckland
New Zealand

fax:64-9-358-1658

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.

PHONE BOOK

The following is a description of IntuiTools the user interface editor that comes with Blitz 2 and then a tutorial to set up a simple phonebook data base system.
IntuiTools is a utility for designing and creating user interfaces for Blitz 2 applications quickly and easily. Full control of Screen, Window and Gadget parameters let you interactively build and test front ends (user interfaces) as complex as you wish.
To begin with we shall run through each of the menu items and their use, then cover some of IntuiTools quirks and lastly, a tutorial which will lead you through the steps needed to create an interface with IntuiTools.

The Menus

PROJECT MENU

LOAD lets you edit previously saved interfaces.
SAVE stores the interface to disk so you can redesign/ edit it later.
CREATE SOURCE generates Blitz 2 code to be inserted into your program.
QUIT is something you contemplate when unhappy with your job.

SCREEN MENU

PALETTE lets you adjust the screen colours used by your program
OPTIONS enables you to change the resolution and other screen attributes

WINDOW MENU

ADD TEXT lets you place text anywhere in your window.
TEST lets you play with your interface just as if your program was running, click in the top left window to exit test mode.
OPTIONS lets you adjust window options as detailed in the Blitz 2 reference manual.

GADGETS MENU

TEXT creates a button with writing on it
STRING creates a string gadget which enables the user to entere text via the keyboard.
PROP is a slider gadget that enables the user to adjust variables accurately.
SHAPE creates a button with a graphic on it, the graphic should be an IFF brush (as created by DPaint etc) and be in the same resolution as your screen.

Positioning Gadgets

After you have entered the relevant information for a gadget a second window appears. You may select whether the gadget’s position is is relative to the top or bottom and left or right of the window. This is only important when the window is sizable.
If the gadget is relative to the bottom right, sizing the window will mean the gadget is repositioned to the new bottom right of the window.
After clicking on MAKE you must position the gadget in your window with the left mouse button.

Gadget Numbering

Each gadget you add to your window needs a specific number known as it’s ID.
No two gadgets in a window can share the same ID.
If your application will have more than one window it is a good idea to use different ID numbers for each window’s gadgets also. However EventWindow can be used to

differentiate gadgets if each window has a gadget with the same ID.
Sometimes leaving some spare ID numbers between groups of gadgets is a good idea if you are to update the interface later. If you have 4 prop gadgets and then a set of 6 text gadgets start the text IDs at 10 so that if you ever need to add another prop gadget to the set you can number them 5,6..
Confused?
Don't worry read this section again after you've done the tutorial.

Paste Mode

Once you have added some gadgets to your window you may wish to copy, delete, move or duplicate them. By clicking on the gadget once you will 'pick it up'.
To delete the gadget just pick it up and then click the right button, the gadget will disappear.
To move the gadget pick it up, move the mouse to the desired position and click the left button to paste it. Then click the right button to exit paste mode.
To duplicate a gadget pick it up, position and click the left button as many times as you want and then hit the right button to exit paste mode.

Tutorial

Before trying this tutorial it's a good idea to re-familiarise yourself with screens, windows and gadgets by reading these chapters in the Blitz reference manual.
First, a hires screen...
Select the Screen/Options menu. Click in the gadget to the right of TITLE:. This is a string gadget. Delete the contents by selecting Amiga X which is a keyboard shortcut to empty the contents of a string gadget. Now type in the name of your application and hit return. PHONE BOOK will do for this application.
Click on the gadget to the right of mode until it reads HiresNonInterlace. This is the typical resolution for an Amiga application. A depth of 2 gives your screen 4 colours. Height should be set to 256 for PAL/European markets and 200 for NTSC/American markets. Because this application will be using the workbench screen we need to set hires with a depth of 2.
When you select DONE the IntuiTools will reconfigure the display for the new settings.
Next, a draggable window and some gadgets...
Now drag the window so it takes up about 1/4 of the whole screen.
Note: to use the sizing gadget in IntuiTools you need to click on the very bottom right of the window.
First we need to add some text to our window that will explain what our string gadgets represent. Select the Window/Add Text menu and type Name then click on MAKE. Position the text in the top left. Add 'Address' and 'Phone' under name.
Now select the Gadget/String menu. Click on Make then Make. Position the box to the right of the 'Name:' text and holding the left mouse button drag the box across to just before the right edge of the window. We now have a string gadget that the user can type in their name, it's ID should be 1.
Using paste mode we need to add 3 more gadgets the same as our name string gadget. Pick it up with the left mousebutton, paste it straight back down again with the left button then paste 3 more times below. Then click on the right mouse button to exit paste mode.
Four gadgets and 3 titles? Thats so we can have two lines for address, using paste mode move the text blocks so they align properly with the gadgets. Click once to pick them up, click again to put them down in their new position and then click the right button to exit paste mode.
Right now for some control gadgets

for our phone book. Using the Gadgets/Text menu option type ’<<‘ in the top field. Then click on MAKE.Type 10 into the gadget ID. Starting our control gadgets at 10 separates them from the string gadgets. Now click on MAKE and add our rewind button to the bottom left.
Use the same procedure as above to add a forward gadget, a ‘PRINT’ gadget and a ’DIAL‘ gadget. Now select SAVE to save our interface as "phonebook.int", and then select CREATE SOURCE to generate some Blitz 2 source code use a filename such as "ram:temp". The Listing is the data base program that uses the interface we have just designed.

```
;
; phone book program by simon
;

FindScreen 0

;the following should be imported from the file ram:t as created in the tutorial

Borders On:BorderPens 1,2:Borders 4,2
StringGadget 0,72,12,0,1,40,239
StringGadget 0,72,27,0,2,40,239
StringGadget 0,72,43,0,3,40,239
StringGadget 0,72,59,0,4,40,239
GadgetJam 0:GadgetPens 1,0
TextGadget 0,8,75,0,10,"NEW ENTRY"
TextGadget 0,97,75,0,11,"| <"
TextGadget 0,129,75,0,12,"<<"
TextGadget 0,161,75,0,13,">>"
TextGadget 0,193,75,0,14,">| "
TextGadget 0,226,75,0,15,"DIAL"
TextGadget 0,270,75,0,16,"LABEL"

SizeLimits 32,32,-1,-1
Window 0,0,24,331,91,$100E,"MY PHONE BOOK",1,2,0
WLocate 2,19:WJam 0:WColour 1,0
Print "Address"
WLocate 19,50
Print "Phone"
WLocate 27,3
Print "Name"

; and nowwe start typing...

#num=4      ;4 strings for each person

NEWTYPE .person
 info$[#num]
End NEWTYPE

Dim List people.person(200)

USEPATH people()
```

```
If ReadFile (0,"phonebook.data")   ;read in names etc from sequential file
        FileInput 0
        While NOT Eof(0)
                If AddItem(people())
                        For i=0 To #num-1:\info[i]=Edit$(128):Next
                EndIf
        Wend
EndIf

ResetList people()

If NOT NextItem(people()) Then AddItem people() ;if empty add blank record

refresh:
        ref=0
        For i=0 To #num-1:SetString 0,i+1,\info[i]:Redraw 0,i+1:Next
        ActivateString 0,1:VWait 5
        Repeat
                ev.l=WaitEvent
                ;
                If ev=$200  ;close window event
                        Gosub update
                        If WriteFile (0,"phonebook.data")  ;save data to file
                                FileOutput 0
                                ResetList people()
                                While NextItem(people())
                                For i=0 To #num-1:NPrint \info[i]:Next
                                Wend
                                CloseFile 0
                        EndIf
                        End
                EndIf
                ;
                If ev=64
                        If GadgetHit=#num Then ActivateString 0,1
                        If GadgetHit<#num Then ActivateString 0,GadgetHit+1
                        Select GadgetHit
                                Case 10:Gosub update:If AddItem(people()) Then ref=1
                                Case 11:Gosub update:If FirstItem(people()) Then ref=1
                                Case 12:Gosub update:If PrevItem(people()) Then ref=1
                                Case 13:Gosub update:If NextItem(people()) Then ref=1
                                Case 14:Gosub update:If LastItem(people()) Then ref=1
                        End Select
                EndIf
        Until ref=1
        Goto refresh

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

;PhoneBook Listing

Additional Commands

Statement: **Block**

Syntax: **Block** *Shape#,X,Y*

Modes: Amiga/Blitz

Description:

Block is an extremely fast version of the Blit command with some restrictions. Block should only be used with shapes that are 16,32,48,64... pixels wide and that are being blitted to an x position of 0,16,32,48,64... Note that the height and y destination of the shape are not limited by the Block command.

Block is intended for use with map type displays.

Statement: **LoadFont**

Syntax: **LoadFont** *IntuiFont#,Fontname.font$,Y Size*

Modes: Amiga

Description:

LoadFont is used to load a font from the fonts: directory. Unlike BlitzFonts any size IntuiFont can be used. The command WindowFont is used to set text output to a certain IntuiFont in a particular Window.

Function: **VPos** (add to chapter 5)

Syntax: **VPos**

Modes: Amiga/Blitz

Description:

VPos returns the video's beam vertical position. Useful in both highspeed animation where screen update may need to be synced to a certain video beam position (not just the top of frame as with **VWait**) and for a fast random nember generator in non frame-synced applications.

The 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

;an 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#)*

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 so 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

SetVoice alters 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!

```
;
; 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 !"
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**

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| #RXFF_RESULT| 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**

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()

AGA PALETTE HANDLING

Blitz 2’s palette object has (again) changed. Palette objects are now capable of containing AGA compatible 24 bit colours.

The new palette objects look likethis:

```
NEWTYPE.rgbcomp
   _red.l              ;left justified red component.
   _green.l            ;left justified green component.
   _blue.l             ;left justified blue component.
End NEWTYPE

NEWTYPE.palettedata
   _numcols.w          ;same as palette/_numcols.
   _zero.w             ;for compatibility with graphics lib
                       ;LoadRGB32.
   _rgbs.rgbcomp(256)  ;256 is the max the amount will actually
                       ;depend upon the highest palette entry.
   _zero2.l            ;for graphics lib too.
End NEWTYPE
```

This is the actual object return by Addr Palette(n):

```
NEWTYPE.palette
   _*data.palettedata  ;00: NULL if no palette present
                       ;    else a pointer to  palettedata.
   _numcols.w          ;04: num cols present in palettedata.
                       ;     below is colour cycling info.
   _lowcol.w           ;06: low colour for cycle range.
   _hicol.w            ;08: high colour for cycle range.
   _speed.w            ;10: speed of cycle : 16384 = max speed
                       ;  sign  indicates cycling direction.
   _var.w              ;12: cvariable speed is added to.
                       ;
                       ; more possible cycling entries....
                       ;
                       ;128: sizeof.
End NEWTYPE
```

Now for the new AGA functions added to Blitz 2...these will all generate a runtime error if used on a non-AGA Amiga....

Statement: **AGARGB**

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

Modes: Amiga

Description:

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

Example:

```
;
; AGA test
;

Screen 0,0,0,1280,512,8,$8024,"SUPER HIRES 256 COLORS",1,2

ScreensBitMap 0,0

For i=0 To 255
    AGARGB i,i/2,i/3,i       ;shades of purple
    Circle 640,256,i*2,i,i   ;big SMOOTH circles
Next

MouseWait
```

Statement: **AGAPalRGB**

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

Modes: Amiga

Description:

The AGAPalRGB command is the AGA equivalent of the PalRGB command. AGAPalRGB allows you to set an individual colour register within a palette object. This command only sets up an entry in a palette object, and will not alter the actual screen palette until a 'Use Palette' is executed.

Function: **AGARed**

Syntax: **AGARed***(colour register)*

Modes: Amiga

Description:

The AGARed function returns the red component of the specified colour register within the currently used screen. The returned value will be within the range 0 (being no red) through 255 (being full red).

Function: **AGAGreen**

Syntax: **AGAGreen***(colour register)*

Modes: Amiga

Description:

The AGAGreen function returns the green component of the specified colour register within the currently used screen. The returned value will be within the range 0 (being no green) through 255 (being full green).

Function: **AGABlue**

Syntax: **AGABlue***(colour register)*

Modes: Amiga

Description:

The AGABlue function returns the blue component of the specified colour register within the currently used screen. The returned value will be within the range 0 (being no blue) through 255 (being full blue).

NEW SCREEN FLAGS

The superhires viewmode flag $20 is now acceptable, but should always be used in conjunction with the standard hires flag of $8000.

The depth of a screen may now be specified up to 8 bitplanes (256 colours) deep (if you've got an AGA machine!). Here's how you would go about opening a superhires, 256 colour screen:

Screen 0,0,0,1280,256,8,$8020,"MyScreen",1,0

3.0 BITMAP HANDLING

Blitz 2's Bitmap object has been upgraded to allow for interleaved bitmaps:

```
NewType.Bitmap
_     ebwidth[0]          ;00: for compatability.
_     linemod.w           ;00: value to get from one scanline to next.
_     height.w            ;02: currently pixel height - but open to commodore
                          ;'enhancement'.
_     depth.w             ;04: number of bitplanes.
_     pad.b[2]            ;06: nothing.
_     data.l[8]           ;08: actual bitplane pointers.
_     pad2.b[12]          ;40: zilch.
_     flags.w             ;0=normal bitmap, <0=interleaved.
_     bitplanemod.w        ;value to get from one bitplane to next. MAY BE 0!
_     xclip.w             ;pixel width for render clipping
_     yclip.w             ;pixel height for render clipping
_     cclip.w             ;number of colours available on bitmap ( = 2^_depth)
_     isreal.w            ;0=no bitmap here, <0=blitz created bitmap, >0=borrowed
                          ;64: sizeof
End NEWTYPE
```

Also, many Blitz2 bitmap related commands have been altered to take this new object into account.

NEW GADGET HANDLING

A new bit, bit 9, in the 'Flags' parameter of the 'TextGadget' and 'ShapeGadget' commands allow you to create mutually exclusive radio button type gadgets. These gadgets DO NOT require Kickstart 2.0 to operate!

Here is an example of setting up some radio button style text gadgets:

```
TextGadget 0,16,16,512,1,"OPTION 1":Toggle 0,1,On
TextGadget 0,16,32,512,2,"OPTION 2"
TextGadget 0,16,48,512,3,"OPTION 3"
```

The new 'ButtonGroup' command allows you to specify which 'group' a series of button gadgets belong to. See 'ButtonGadget' below.

Note that if you are using button gadgets, you SHOULD really toggle ONE of the gadgets 'On' before giving the gadgetlist to a window - as in the example above.

Text Gadgets may now be used to create 'cycling' gadgets. Again, these gadgets DO NOT require kickstart 2.0 to work.

If you create a text gadget which contains the '| ' character in the gadget's text, Blitz 2 will recognize this as a 'cycling' gadget, using the '| ' character to separate the options - like this:

```
        TextGadget 0,16,16,0,1," HELLO | GOODBYE|  SEEYA | "
```

Now, each time this gadget is clicked on, the gadgets text will cycle through ’Hello’, ’GOODBYE’ and ’SEEYA’. Note that each option is spaced out to be of equal length. This feature should not be used with a GadgetJam mode of 0.

Function: **GadgetStatus**

Syntax: **GadgetStatus***(GadgetList#,Id)*

Modes: Amiga

Description:

GadgetStatus may be used to determine the status of the specified gadget. In the case of ’toggle’ type gadget, GadgetStatus will return true (-1) if the gadget is currently on, or false (0) if the gadget is currently off.

In the case of a cycling text gadget, GadgetStatus will return a value of 1 or greater representing the currently displayed text within the gadget.

Statement: **ButtonGroup**

Syntax: **ButtonGroup** *Group*

Modes: Amiga

Description:

ButtonGroup allows you to determine which ’group’ a number of button type gadgets belong to. Following the execution of ButtonGroup, any button gadgets created will be identified as belonging to the specified group. The upshot of all this is that button gadgets are only mutually exclusive to other button gadgets within the same group.

’Group’ must be a positive number greater than 0. Any button gadgets created before a ’ButtonGroup’ command is executed will belong to group 1.

Function: **ButtonId**

Syntax: **ButtonId***(GadgetList#,ButtonGroup)*

Modes: Amiga

Description:

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

Statements: **Enable & Disable**

Syntax: **Enable** *GadgetList#,Id* & **Disable** *GadgetList#,Id*

Modes: Amiga

Description:

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

Statement: **SetGadgetStatus**

Syntax: **SetGadgetStatus** *GadgetList#,Id,Value*

Modes: Amiga

Description:

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

NEW GADGETS EXAMPLE:

```
;
; new gadget types
;
WBStartup:FindScreen 0     ;open on workbench
TextGadget 0,32,14,0,0,"CYCLE 1|CYCLE 2|CYCLE 3"

ButtonGroup 1  ;first group of radio buttons follows
For i=1 To 5
    TextGadget 0,32,14+i*14,512,i,"CHANNEL #"+Str$(i)
Next

ButtonGroup 2  ;second group of radio buttons follows
For i=6 To 10
    TextGadget 0,32,14+i*14,512,i,"BAND #"+Str$(i)
Next
Window 0,20,20,160,180,$1008,"GADGET TEST",1,2,0

Repeat            ;wait until close window gadget hit
    ev.l=WaitEvent
Until ev=$200
```

DATE & TIME COMMANDS

Function: **SystemDate**

Syntax: **SystemDate**

Modes: Amiga

Description:

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

Example:

```
;
; date/time test
;

Dim d$(6):Restore daynames:For i=0 To 6:Read d$(i):Next
Dim m$(12):Restore monthnames:For i=1 To 12:Read m$(i):Next

NPrint Date$(SystemDate)
NPrint d$(WeekDay)," ",Days," ",m$(Months)," ",Years
NPrint Hours,":",Mins,":",Secs
NPrint "press mouse to quit"
MouseWait

daynames:
   Data$ SUNDAY,MONDAY,TUESDAY,WEDNESDAY
   Data$ THURSDAY,FRIDAY,SATURDAY
monthnames:
   Data$ JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC
```

Function: **Date$**

Syntax: **Date$** (days)

Modes: Amiga

Description:

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

Function: **NumDays**

Syntax: **NumDays** (date$)

Modes: Amiga

Description:

Numdays converts a Date$ in the above format to the day count format, where numdays is the number of days since 1/1/1978.

Statement: **DateFormat**

Syntax: **DateFormat** format# ; 0 or 1

Modes: Amiga

Description:

DateFormat configures the way both date$ and numdays treat a string representation of the date: 0=dd/mm/yyyy and 1=mm/dd/yyyy

Functions: **Days Months Years & WeekDay**

Syntax: **Days Months Years & WeekDay**

Modes: Amiga

Description:

Days Months and Years each return the particular value relevant to the last call to **SystemDate**. They are most useful for when the program needs to format the output of the date other than that produced by date$. WeekDay returns which day of the week it is with Sunday=0 through to Saturday=6.

Functions: **Hours Mins & Secs**

Syntax: **Hours Mins & Secs**

Modes: Amiga

Description:

Hours, Mins and Secs return the time of day when **SystemDate** was last called.

ENVIRONMENT COMMANDS

Functions: **WBWidth Height Depth & ViewMode**

Syntax: **WBWidth, WBHeight, WBDepth & WBViewMode**

Modes: Amiga

Description:

The functions WBWidth, WBHeight, WBDepth & WBViewMode return the width, height, depth & viewmode of the current WorkBench screen as configured by preferences.

Functions: **Processor & ExecVersion**

Syntax: **Processor & ExecVersion**

Modes: Amiga

Description:

The two functions Processor & ExecVersion return the relevant information about the system the program is running on. The values returned are as follows:

ExecVersion	OS Release
33	1.2
34?	1.3
36	2.0
39	3.0

Processor	Part#
0	68000
1	68010
2	68020
3	68030
4	68040

NEW DRAWING COMMANDS

Statement: **Poly & Polyf**

Syntax:
Poly numpoints,*coords.w,color & **Polyf** numpoints,*coords.w,color[,color2]

Modes: Amiga/Blitz

Description:

Poly & **Polyf** are bitmap based commands such as **Box** and **Line**. They draw polygons (unfilled and filled respectively) using coordinates from an array or newtype of words. Polyf has an optional parameter color2, if used this colour will be used if the coordinates are listed in anti-clockwise order, useful for 3D type applications. If color2= -1 then the polygon is not drawn if the verticies are listed in anti-clockwise order.

Example:

```
NEWTYPE .tri:x0.w:y0:x1:y1:x2:y2:End NEWTYPE
BLITZ
BitMap 0,320,256,3
Slice 0,44,3:Show 0
While Joyb(0)=0
  a.tri\x0=Rnd(320),Rnd(256),Rnd(320),Rnd(256),Rnd(320),Rnd(256)
  Polyf 3,a,1+Rnd(7)
Wend
```

Statement: **BitPlanesBitMap**

Syntax: **BitPlanesBitMap** *SrcBitMap, DestBitMap, PlanePick*

Modes: Amiga/Blitz

BitPlanesBitMap creates a ‘dummy’ bitmap from the SrcBitMap with only the bitplanes specified by the PlanePick mask. This is useful for shadow effects etc. where blitting speed can be speed up because of the fewer bitplanes involved

Statement: **ClipBlit**

Syntax: **ClipBlit** *Shape#,X,Y*

Modes: Amiga/Blitz

ClipBlit is the same as the **Blit** command except **ClipBlit** will clip the shape to the inside of the used bitmap, all blit commands in Blitz2 are due to be expanded with this feature.