DrawTools 3.1

by Ken O. Burtch

For questions or comments, please write to the following address:

Pegasoft
Honsberger Avenue, R.R.#1
Jordan Station, Ontario, Canada
L0R 1S0

CINDY & MARSHALL BROWN
c/o SAUDI ARAMCO BOX 1030
DHAHRAN 31311
KINGDOM OF SAUDI ARABIA

Table of Contents

I. User Guide

1. Introduction

1.1. Introduction & Legal Stuff

The part of this manual is a general introduction to DrawTools. It isn't a tutorial on computer graphics, although some basic topics are discussed. For more in depth information on specific tools, consult the reference section.

We'd love to hear from you. If you have any questions, comments, or complaints, please feel free to write to Pegasoft at:

> Pegasoft
> Honsberger Avenue, R. R. #1
> Jordan Station, Ontario, Canada
> L0R 1S0

This manual and the related software contained on the diskettes are copyrighted materials. All rights reserved. Duplication of any of the above described materials, for other than personal use of the purchaser, without express written permission of Pegasoft of Canada is a violation of the copyright law of the United States and Canada, and is subject to both civil and criminal prosecution.

Pegasoft and DrawTools are trademarks of Pegasoft of Canada.

1.2. What is DrawTools?

Welcome to DrawTools, a collection of over 100 useful graphics and animation tools for the IIGS. The first version was released as shareware around the fall of 1990. Since then, it has significantly grown, with new features and more versatility.

Feel free to distribute the TOOL098 file with any programs you make, but if you wish to distribute any other files on the DrawTools disks, please get prior permission from Pegasoft.

1.3. System Requirements

DrawTools 3.1 requires the following:

An Apple IIGS with system software 5.0.2 and at least 9K free RAM in bank 0.
To use DrawTools, the following toolsets must be active: Tool Locator, Misc. Tools, Memory Manager, QuickDraw II.

1.4. Installation

1. Copy the TOOL098 file to the Tools folder of your startup disk. (This is DrawTools.)
2. Copy the DT.Drivers folder (the folder and its contents) to the System:Tools folder of your startup disk.
3. Copy the icon file to your Icons folder.

The DrawTools disks also contain the following:
> a. PicEd 3.0, a simple editor for picture libraries
> b. Lib.Converter 1.2, a utility which translates a screen template into a picture library. The folder includes some sample templates.
> c. Demo.Game, a small assembly language game that demonstrate some of the animation tools

d. Demo.Sys16, a demo program written in Micol Advanced Basic 4.2

e. sample programs for a wide variety of computer languages

1.5. Using DrawTools with ...

Complete/TML Pascal II - an interface file written in TML Pascal II is included on the disk in the TML.Pascal folder. Copy the object file to the folder containing the interface files for the other toolsets. Include DrawTools in your USES list at the beginning of your program.

Micol Advanced BASIC - You need to use the TOOLBOX command. A set of aliases are supplied for users with the latest version of BASIC: you can copy these into your program or you can use the INCLUDE command. Each alias requires a space after the tool name.

DrawShadow will not work unless you are running a stand-alone application. There are also some tools that require a Pascal string (not a BASIC string): a length (byte) followed by the text of the string. You cannot use these tools directly: you will either have to construct a string with POKEs, or use Micol Macro. All the toolsets that DrawTools requires are started for you when you use HGR or HGR2.

Merlin 16+ - a macro file (Draw.Macs.S) is included on the disk. Copy it into your MACRO.LIBRARY subdirectory, and USE it in your source files.

ORCA/Pascal - an interface file (Drawtools.int) is included on the disk. Copy this file into OrcaPascalDefs. In your program, include DrawTools in your USES list.

ORCA/M - A macro file (m16.DrawTools) is included on the disk. Copy this file into your ainclude folder. Use it like any other macro file.

ORCA/C - There are no interface files available: you can use DrawTools if you use the necessary tool definitions.

DrawTools will not work with **Prizm**.

Pegasus Pascal - Follow the ORCA/Pascal directions.

Example: Starting DrawTools 3.1.

Pegasus Pascal: Start it like any other toolset.

```
Uses Common, ..., DrawTools
| start required tools, or use StartGraphics
LoadOneTool 98, 0                             | Load DrawTools
DPHandle = NewHandle(256, MyID, $C005, 0)     | allocate direct page space
DP = ord( DPHandle^ )                          | convert to an integer
DrawStartUp DP, MyID                           | start DrawTools
ExtendBuffers                                  | if using a lot of pixies
```

ORCA/Pascal: Start it like any other toolset.

```
Uses Common, ..., DrawTools;
{ start required tools, or use StartGraphics }
LoadOneTool(98, 0);                           { Load DrawTools }
DPHandle := NewHandle(256, MyID, $C005, 0);   { allocate direct page space }
DP := ord( DPHandle^ );                        { convert to an integer }
DrawStartUp( DP, MyID );                        { start DrawTools }
ExtendBuffers;                                  { if using a lot of pixies }
```

BASIC: Use the following commands:

```
REM Start required tools, or use HGR/HGR2.
TOOLBOX(1, 15; 98, 0)  : REM Tool Locator's LoadOneTool
DrawTools_Handle = 256  : REM Allocate direct page space
DrawTools_Address = 0
POKE 202, 1
Get_Mem( DrawTools_Handle, DrawTools_Address)
Address% = INT(DrawTools_Address): REM Convert to an integer
MyID% = Peek(238) + Peek(239) * 256
TOOLBOX( ~DrawStartUp : Address%, MyID%)
REM Without aliases: TOOLBOX( 98, 2 : Address%, MyID% )
TOOLBOX( ~ExtendBuffers )  : REM If using a lot of pixies
```

Merlin 16+: Start it like any other toolset.

```
USE 4:Draw.Macs
~LoadOneTool #98;#0
~NewHandle #$100;MyID;#$C005;#0
PLA
PushWord MyID
_DrawStartUp
_ExtendBuffers                                    ; if using a lot of pixies
```

ORCA/M: Start it like any other toolset.

```
MCOPY m16.DrawTools
ph2 #98       ; load DrawTools
ph2 #0
_LoadOneTool
ph4 #0
ph4 #$100
ph2 MyID
ph2 #$C005
ph4 #0
_NewHandle
pha                                              ;
ph2  MyID
_DrawStartUp
_ExtendBuffers                                    ; if using a lot of pixies
```

Examples: Stopping DrawTools 3.1.
Pegasus Pascal:

```
DrawShutDown
```

ORCA/Pascal:

```
DrawShutDown;
```

BASIC:

```
TOOLBOX(~DrawShutDown )
REM Without aliases: TOOLBOX(98, 3)
```

Merlin 16+:

```
_DrawShutDown
```

ORCA/M:

```
_DrawShutDown
```

2. Graphics on the IIGS

2.1. A Brief Introduction

Graphics is the art of drawing with a computer. In the IIGS, there is a special toolset dedicated to drawing called "QuickDraw II", or QuickDraw for short. QuickDraw provides all the basic drawing functions for the average application: it draws lines, rectangles, ovals, text, cursors and many other things you see on the screen. It's impossible to make a complete list of the QuickDraw tools here since there are well over 200; consult the Apple IIGS Toolbox Reference or any book introducing IIGS programming for more information.

[P] BASIC: Whenever you use HCOLOR, HPLOT, or the other BASIC commands, BASIC uses QuickDraw.

Before discussing the details of DrawTools, you should know a little bit of how pictures are displayed on the IIGS screen. We will be discussing 320 mode to keep things simple. The super high-resolution graphics screen is located in bank $E1 of memory. Each dot on the screen, or "pixel", consists of half a byte of memory , or 4 bits. This means up to sixteen colours can normally be displayed on the screen. The screen consists of 320 pixels horizontally and 200 pixels vertically. These pixels are located in the area $E12000 to $E19CFF of memory.

The next 200 bytes, starting at $E19D00, are for the Scanline Control Bytes, or SCB's, one for each line on the screen. The SCB's determine the attributes for that line:

> bit 0...3 - the palette of the line (0 to 15)
> bit 4 - zero
> bit 5 - 1 if fill mode is active. With fill mode active, colour 0 (usually black) behaves differently.
>   If you draw an area of the screen in colour 0, it will appear in the same colour as the area of the screen to the immediate left. The colour is "pulled" across the black areas of the screen, filling them in.
> bit 6 - 1 will cause an SCB interrupt on this line
> bit 7 - 1 for 640 resolution; 0 for 320 resolution

The memory located from $E19E00 to $E19FFF contains the 16 colour palettes (or "color tables"). Each palette contains 16 integer RGB values that describe the 16 colours you can see on the screen. QuickDraw only uses palette 0 (see Figure 1). Palettes and RGB colour words are discussed more below.

#	Name	RGB	#	Name	RGB
0	black	$000	8	flesh pink	$FA9
1	dark grey	$777	9	yellow	$FF0
2	brown	$841	10($A)	green	$0E0
3	purple	$72C	11 ($B)	light blue	$4DF
4	blue	$00F	12 ($C)	lilac purple	$DAF
5	dark green	$080	13 ($D)	periwinkle blue (desktop)	$78F
6	orange	$7F0	14 ($E)	light grey	$CCC
7	red	$D00	15 ($F)	white	$FFF

Figure 1 : The QuickDraw II colours (in 320 mode)

This whole section of memory, from $E12000 to $E19FFF, can be "shadowed" from $012000 to $019FFF in bank 1. This area is called the shadow screen. You can use the shadow screen if you set bit 15 in the Master SCB when you start QuickDraw up. When the shadow screen is in use, drawing takes place much faster than usual. In addition, the shadow screen can be made invisible (with DrawTools ShadowOff) so that QuickDraw & DrawTools

draw many times faster than without shadowing, but the pictures will remain hidden until use DrawTools' QuickWipe.

2.2 Working with Colour

On the IIGS, the super hires screen can display 16 colours at a time with a single palette. You can change the current drawing colour using QuickDraw's SetSolidPenPat(c) or BASIC's HCOLOR=c. The hue and brightness of each colour is described by an RGB colour word, a combination of red, green and blue components. Each component can be in a range from 0 to 15. For example, black is 0,0,0; white is 15,15,15; bright red is 15,0,0; orange is 15,7,0.

DrawTools has a tool called **SetColour** that will take the red, green and blue components and give you the corresponding RGB value.

<u>Example:</u> Creating the colour "orange" with SetColour.
Pascal: RGBColour := SetColour(15, 7, 0);
BASIC: TOOLBOX(~SetColour : 0, 15, 7, 0; RGBColour%)
        REM Include 0 at start for RGBColour%! Add one 0 for each result value.

<u>Example:</u> You can use QuickDraw's SetColorEntry to change a default colour:
Pascal: SetColorEntry(0, 5, SetColour(15, 7, 0));
BASIC: TOOLBOX(~SetColour : 0, 15, 7, 0; RGBColour%)
        TOOLBOX(4, 16: 0, 0, RGBColour%)

Besides SetColour, there is a **SetColPercent** will do the same thing, accept you use percentages (0...100) of red, green and blue components instead of values from 0...15. **FadeColour** will make an RGB value darker or brighter. **BlendColour** will blend to colours together to make a new colour. **FindColour** will find the closest colour in a palette to the colour word you specify.

Although QuickDraw uses one palette, the IIGS can actually display colours from 16 different palettes at one time. Each line must have only one palette. DrawTools has a tool called **SetPalette** that will change the palette for a set of lines.

<u>Example:</u> Changing the top half of the screen (lines 0...99) to palette 1.
Pascal: SetPalette(0, 99, 1);
BASIC: TOOLBOX(~SetPalette : 0, 99, 1)

Now anything you draw on the top half of the screen will appear in the colours of palette 1 instead of palette 0. You can set the colours of any of the palettes using SetColorEntry(palette, colour, RGBvalue); or in BASIC, TOOLBOX(4, 16: palette%, colour%, RGBvalue%).

FadePal will make make all the colours in a palette darker. **UnfadePal** will make all the colours in a palette brighter. A more powerful version of FindColour is **FindPalette**. Give FindPalette a palette of colours, and it will try to match them up to colours in the current palette. This tool is useful for NDAs: you can never be sure which colours are on the screen if an NDA is running under a paint program. FindPalette can tell you if the colours have changed, and to what.

<u>Example:</u> See the reference for more details. If you want to find the closest colours on the screen to the standard 320 palette:
   Define the colours array:0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
   Define the Palette:$000, $777, $841, $72C, ..., $78F, $CCC, $FFF
   After the call is made, the values in colour list will change to reflect the actual numbers for these colours on the current screen (or the closest colours them).

2.3 Fades, Wipes, and Dissolves

What set of tools would be complete without some way to gracefully change from one scene to another? There are four basic ways to make such a transition. The simplest way is to erase the screen and draw a new picture; it's easy, effective, but it lacks a certain class, especially on a computer with the possiblities of the IIGS. A common way to switch pictures is with a fade. A fade changes all the colours to a single colour, and then reverses the process to reveal a new picture. While the colours are identical, any drawing you do is invisible. DrawTools provides two fades: (1) QuickFade, the standard fade used in so many applications, which dims all the colours in the first eight palettes to black; (2) IncrFade, which fades out everything except the red component, and then fades to black

A second method of switching pictures is with a wipe. A wipe copies a picture from shadow screen onto the screen in a special order. DrawTools provides two wipes which copy the shadow screen to the main screen: (1) QuickWipe, which instantly copies one to the other, and (2) VBWipe which copies using a Vencian blind effect.

The last way to change screens is a dissolve. This is a special kind of wipe which operates on a pixel-by-pixel basis. There are no dissolves in DrawTools.

Example: How to fade to black, draw something new, and unfade to reveal it.
Pascal:
```
QuickFadeOut(1);
repeat until FadeDone;
{draw the new screen here}
QuickFadeIn(1);
repeat until FadeDone;
```

BASIC:
```
TOOLBOX(~QuickFadeOut : 1)
REPEAT
    TOOLBOX(~FadeDone : 0; FadeDone%)
UNTIL FadeDone% <> 0
REM Draw the new screen here
TOOLBOX(~QuickFadeIn : 1)
REPEAT
    TOOLBOX(~FadeDone : 0; FadeDone%)
UNTIL FadeDone% <> 0
```

Merlin 16+:
```
          ~QuickFadeOut #1
QFOLoop   ~FadeDone
          PLA
          BEQ QFOLoop
* Draw new screen here
          ~QuickFadeIn #1
QFILoop   ~FadeDone
          PLA
          BEQ QFILoop
```

Example: How to use the Venician Blind wipe tool to wipe a new screen over an old one.
Pascal:
```
{make sure the shadow screen is allocated}
DrawShadow;
ShadowOff;
{draw the new screen here}
VBWipe;
```

```
DrawMain; { or ShadowOn, if you want to use the shadow screen }
```

BASIC: Reminder: Uses the shadow screen: stand-alone programs only!

```
TOOLBOX(~DrawShadow )
TOOLBOX(~ShadowOff )
REM Draw new screen here
TOOLBOX(~VBWipe )
TOOLBOX(~DrawMain )
```

Merlin 16+

```
_DrawShadow
_ShadowOff
;draw the new screen here
_VBWipe
_DrawMain
```

3. Animation

3.1 What is Animation?

Animation is the illusion of motion created when a sequence of pictures is rapidly displayed. Each picture, called a cell or frame, is a modified version of the picture before it. When each of these still frames is displayed quickly, one after another, they give the illusion of smooth motion. A movie displays 24 frames per second, and at 60 or beyond the eye can't distinguish animation from real motion. Reasonable computer animation can be achieved at speeds of even 4 frames per second.

To show a flag being raised, you would first start with a picture of flag pole. Then you would create a series of pictures, each with the flag a little farther up the pole. The final picture would be drawn with the flag at the top of the pole. Each of these pictures of the flag and the flag pole is a frame. When you display these pictures rapidly and in order, the flag appears to smoothly rise up the pole. This is the fundamental principle of animation.

3.2 Animation Examples: Dialog Ideas

To view some sample animation sequences, start PicEd and load the dialog.pics picture library included in the PicEd folder. This file is an unpacked picture library created from the Dlog.Ideas320 file using the Library Converter utility. Once the file is loaded, try some of the following animation sequences.

Name	Sequence	Speed
1. Note Alert	0,1,2,3,3,3,255,0	4
2. Caution Alert	4,5,6,7,6,5,255,0	2
3. Stop Alert	8,9,10,11,255,0	3
4. Working GS	12,13,14,12,13,14,12,13,14,15,15,15,255,0	5
5. Swap Disks	16,17,18,19,20,21,22,23,255,0	2

To try one of these animations:
1. Click on the ani button.
2. Click on the seq button.
3. Type in the picture sequence, one number at a time.
4. Click on the Go! button.
5. Type in the speed number.
6. To stop the playback, hold down the mouse button.

Try some experiments.

3.3 Colour Cycling

One of the simplest methods of performing animation is colour cycling. It is the process of changing RGB colour words to make objects on the screen appear and disappear. Most paint programs for the IIGS have some kind of colour cycling feature.

To use colour cycling, you draw only one picture, but you paint different frames in different colours. With the flag pole example, you could draw a series of flags up the flag pole, each in a different colour, the lowest flag in colour 1, the second lowest in colour 2, and so on. When you are finished, you have a flag pole full of coloured flags (see Figure 2). If you change all of the colours except colour 1 to black, the only flag that is visible is the one on the bottom of the flag pole. If you change colour 1 to black, and change colour 2 to the colour of the flag, the second flag on the flag pole appears. By cycling through the current palette, making one colour after another visible,

the flag appears to rise up the pole.

Figure 2: Flagpole Example

Colour   1   2   3   4
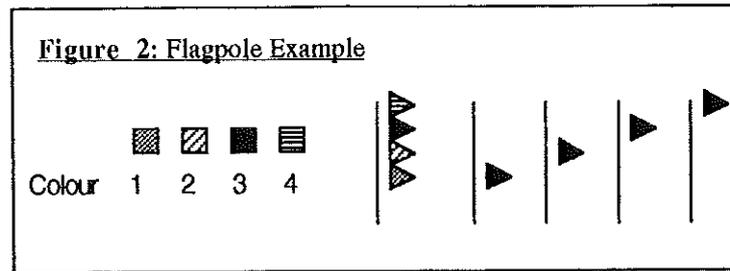
To do colour cycling in the current palette, all you need are two QuickDraw tools: GetColorEntry and SetColorEntry. The first gives you the RGB colour word for a particular palette entry. The second lets you change a colour in a specified position in a palette to a new colour. There is also a GetColorTable and SetColorTable that lets you change whole palettes at a time. Here's an example of how you might write the flag animation:

Example: Colour Cycling of a flag pole with five flags.

Pascal:

```
procedure AnimateFlagPole;
        var OldColours : ColorTable;
        FlagColour, LastFlagColour, i : integer;
begin
        GetColorTable(0, OldColours);{ save the original colours }
        LastFlagColour := 5;{ used to erase old flags }
        for i := 1 to 5 do SetColorEntry(0, i, $000);{ erase all the flags }
        for i := 1 to 100 do begin{ one hundred times }
            for FlagColour := 1 to 5 do begin{ for each flag colour }
                SetColorEntry(0, LastFlagColour, $000);{ Make the last flag invisible }
                SetColorEntry(0, FlagColour, $FFF);{ draw the flag in white }
                LastFlagColour := FlagColour;{ this flag gets erased next }
            end;
        end;
        SetColorTable(0, OldColours);{ restore original colours }
end {AnimateFlagPole};
```

BASIC:

```
DIM OldColours%(15)
...
PROC AnimateFlagPole
        OldColoursL% = ADDR( OldColours%( ) )   : REM Get address of array
        OldColoursH% =PEEK(202)
        TOOLBOX( 4, 15 : 0, OldColoursH%, OldColoursL% )  : REM Save colours in the array
        LastFlagColour% = 5                             : REM Used to erase old flags
        FOR i% = 1 TO 5                                 : REM Erase all the flags
            TOOLBOX( 4, 16: 0, i%, 0 )
        NEXT i%
        FOR i% = 1 TO 100                               : REM One hundred times
            FOR FlagColour% = 1 TO 5                    : REM For each flag colour
                TOOLBOX( 4, 16: 0,LastFlagColour%, 0)   : REM Make the last flag invisible
                TOOLBOX( 4, 16: 0,FlagColour%, 4095)    : REM Draw the flag in white
                LastFlagColour% = FlagColour%           : REM this flag gets erased next
            NEXT FlagColour%
        NEXT i%
```

```
        TOOLBOX(4, 14: 0, OldColoursH%, OldColoursL%)  : REM Restore original colours
ENDPROC
```

You can do even more impressive colour cycling by changing an entire palette at a time. This is used by many video games to create animation across the whole screen without having to do a lot of work. For example, the pixels for water may never be redrawn. Water looks like it's moving because the colours of the water pixels are slowly changing. This is an impressive animation effect that take very little effort on the part of a program.

DrawTools provides two tools for palettes that work like GetColorEntry and SetColorEntry. GetPalette gives you the palette being used for a particular line on the screen. SetPalette, which we have seen before, lets you change the current palette over a range of lines.

Example: Palette Cycling.
Pascal:
```
procedure CyclePalettes;
        var OldPalette, PalNum, Delay, i : integer;
begin
        OldPalette := GetPalette(1);{ save the original palette number }
        for i := 1 to 100 do begin{ one hundred times }
            for PalNum := 0 to 15 do begin{ change the screen palette }
                SetPalette(0, 199, PalNum);{ to each of the 16 palettes }
                for delay := 1 to 5 do WaitVB;{ time delay = 1/6 second }
            end;
        end;
        SetPalette(0, 199, OldPalette);{ restore the original palette }
end {CyclePalettes};
```

BASIC:
```
PROC CyclePalettes
        TOOLBOX(~GetPalette : 0, 1 ; OldPalette%) : REM save the original palette
        FOR i% = 1 TO 100
            FOR PalNum% = 0 TO 15 : REM change the screen palette
                TOOLBOX(~SetPalette :0, 199, PalNum%)
                FOR delay% = 1 TO 5
                    TOOLBOX(~WaitVB ): REM time delay = 1/6 second
                NEXT delay%
            NEXT PalNum%
        NEXT i%
        TOOLBOX(~SetPalette :0, 199, OldPalette%) : REM restore the original palette
ENDPROC
```

3.4 The Art of Animation: Draw, Erase and Redraw

Colour cycling is fine for some kinds of animation, but a program often needs to save many of the colours in the palette for other uses. The more conventional approach to animation is to draw an object on the screen, erase it, and then draw it again somewhere else. This cycle of draw, erase, draw, erase, is the technique used in most computer games.

The one problem with draw/erase/redraw animation is flicker. This occurs when the object being animated can't be redrawn fast enough. The eye sees the picture when it's there and when it isn't there, and this makes the object you're animating appear to flicker. One of the easiest ways of reducing flicker is to use DrawTools' WaitVB tool before you try to erase anything.

Example: The following procedure moves a white box across the screen by drawing it, erasing it, and then redrawing

it. WaitVB is used to keep the flicker low. To see the box flicker, try replacing the WaitVB loop with "for delay :=
1 to 5000 do ;" and change the number of iterations.

Pascal:

```
procedure MoveABox;
        var Box : rect; i, delay : integer;
begin
        SetRect( Box, 0, 10, 30, 40);{ the 30 x 30 box }
        SetSolidPenPat( 15 );{ the box colour }
        SetSolidBackPat( 0 );{ the erasing colour }
        for i := 1 to 20 do begin{ 20 times }
            OffsetRect( Box, 10, 0);{ move the box 10 pixels to the right }
            PaintRect( Box );{ draw it }
            for delay := 1 to 5 do WaitVB;{time delay = 1/6 second }
                                                    { we just finished a WaitVB! }
            EraseRect( Box );{ erase the box without flicker }
        end;
end;
```

BASIC:

```
DIM Box%(8) : REM Space for a rectangle
...
PROC MoveABox
REM I'm assuming BoxH% & BoxL% is the address of the box% array.
        TOOLBOX( 4, 74 : BoxH%, BoxL%, 0, 10, 30, 40) : REM create a 30 x 30 box
        HCOLOR = 15
        BKCOLOR = 0
        FOR i% = 1 TO 20 : REM 20 times
            TOOLBOX(4, 75 : BoxH%, BoxL%, 10, 0) : REM move the box 10 pixels to the right
            TOOLBOX(4, 84 : BoxH%, BoxL%) : REM  draw it
            FOR delay% = 1 TO 5
                TOOLBOX(~WaitVB ) : REM time delay = 1/6 second
            NEXT delay% .
        TOOLBOX(4, 85 : BoxH%, BoxL% ): REM erase the box without flicker!
        END i%
ENDPROC
```

3.5 Bit-Mapped Pictures

Each time QuickDraw paints an object on the screen (like our box) it has to do a number of things:

a) Make sure the mouse arrow isn't erased.
b) Make sure the object is actually on the screen.
c) Make sure the object is within the clipping & visible regions of the current window or grafport.
d) Calculate which colours to use with the pen pattern and pen mask.
e) Compute which pixels to change in the current pen mode & size.

All this is what makes QuickDraw so handy and powerful, but it also makes it slow, too slow except for the
simplest kinds of animation. After all, if we are drawing a space ship, we don't need special pen modes, sizes,
patterns and the rest of those features. To draw a picture very quickly, DrawTools provides a special set of tools call
the drawing tools. There are 8 drawing tools: **Draw, Draw48, DrawAt, Draw48At, DrawOn, Draw48On,
DrawOnAt** and **Draw48OnAt**. The basic tool, Draw, draws a bit-mapped picture library picture (24 pixels wide
and 24 pixels high, the ones used with PicEd and the Library Converter). The other tools are variations on Draw:

the "48" tools draw 4 pictures at once (like you see in the double-sized window in PicEd); the "At" tools let you specify the screen position to draw at; and the "On" tools let you draw matted pictures. We'll talk more about pictures and mattes later. Because each of these tools is customised for a particular size and "pen mode", they draw pictures many times faster than QuickDraw can.

Before we can use the drawing tools, we need to load a picture library from a disk with the LoadLibrary tool. A picture library is a set of 32 bit-mapped pictures created with PicEd or the Library Converter utility. LoadLibrary loads picture library from a disk and it gives you an "ID code" that you can use later on to refer to the library. LoadLibrary has some special parameters that will be described later on when we talk about matting. There is also an UnloadLibrary tool, but you normally don't need to use it.

You can only draw with one library at a time. To specify which library we want to draw with, we need to use the SetLibrary tool. There is also a GetLibrary tool that returns the ID code for the current library.

> LoadLibrary uses a GS/OS string for the pathname: there is no direct way in BASIC to use GS/OS strings. We can fake the LoadLibrary/SetLibrary calls with BLOAD. This only works with unpacked libraries.

Example: Loading a library in BASIC without LoadLibrary or SetLibrary.
BASIC:
```
REM DrawTools_Addr% is the direct page space you allocated when you started DrawTools.
DrawTools_Buffer = PEEK(DrawTools_Addr%+4) + PEEK(DrawTools_Addr%+5) * 256
POKE 202,0 : REM In BASIC 5.0, we what to load the whole library
BLOAD "path name of picture library", DrawTools_Buffer, 9216
```

Example: The following procedure demonstrates how to load and display the pictures in an (unpacked) library. The LoadLibrary tool requires a GS/OS string (a two-byte length followed by the string itself), so refer to your particular language on how to define a GS/OS string.

Pascal:
```
procedure DumpOutLibrary( pathname : GSOSString);
        var TheLibrary : integer;{ ID code for the library }
begin
        TheLibrary := LoadLibrary( pathname, 0, 0);{ load the library from disk }
        SetLibrary( TheLibrary );{ use this library to draw with }
        for y := 0 to 3 do { 4 rows }
            for x := 0 to 7 do { 8 pictures per row }
                DrawAt( x * 32, y * 32, x + y * 8 );{ draw picture # x+8y }
end;
```

BASIC:
```
PROC DumpOutLibrary
REM Fake the LoadLibrary/SetLibrary as described above (or use Micol Macro & LINK).
        FOR y% = 0 TO 3 : REM 4 rows
            FOR x% = 0 TO 7 : REM 8 pictures per row
                ScreenX% = x% * 32
                ScreenY% = y% * 32
                PicNum% = x% + y% * 8
                TOOLBOX(~DrawAt : ScreenX%, ScreenY%, PicNum% )
            NEXT x%
        NEXT y%
ENDPROC
```

Example: You can animate pictures with the drawing tools in the same way that we animated the box. If you create a picture of a box using PicEd in picture 0 of a library, and you leave picture 1 of the library blank (to erase with), then you can animate this box using the following procedure.

Pascal:

```
procedure MoveBoxInADrawToolsLibrary( pathname : GSOSString);
const   Box = 0;{ box is picture zero }
        Blank = 1;{ blank picture is picture 1 }
var     i, delay : integer; BoxLib : integer;
begin
        BoxLib := LoadLibrary( pathname,   0, 0);{ load the pictures }
        SetLibrary( BoxLib );{ draw with this set }
        for i := 1 to 28 do begin{ 28 times }
            DrawAt( i * 10, 20, Box);{ draw a box }
        .   for delay := 1 to 5 do WaitVB;{time delay = 1/6 second }
            DrawAt( i * 10, 20, Blank);{ erase the box }
        end;
end;
```

BASIC:

```
PROC MoveBoxInADrawToolsLibrary
        Box% = 0
        Blank% = 1
        REM Load and set the library
        FOR i% = 1 TO 28
            x% = i% * 10
            TOOLBOX(~DrawAt : x%, 20, Box%)
            FOR delay% = 1 TO 5
                TOOLBOX(~WaitVB )
            NEXT delay%
            TOOLBOX(~DrawAt : x%, 20, Blank%)
        NEXT i%
ENDPROC
```

3.6 Caching with Library Buffers

DrawTools provides a caching mechanism that can reduce the swapping time when you change from one library to another with SetLibrary. If you need the extra speed that caching provides, use the ExtendBuffers tool after DrawStartUp. Now each time you use SetLibrary, the library will be loaded into a library buffer in bank 0. If you use SetLibrary to select a library which is already in a buffer, DrawTools will switch to the appropriate buffer without reloading the library from main memory.

DrawTools can allocate up to 5 library buffers. Only the libraries you use the most will be cached; in order to get the best performance from the caching mechanism, use the ResetBuffers tool when you are about to use a new set of libraries. This clears the old libraries from the library buffers in preparation to receive a new set of libraries, such as when a new level in a game is about to start.

You can pre-load the library buffers when they are clear by using SetLibrary once for each library you will be using.

3.7 Mattes : Merging the Background with a Picture

Using DrawAt, we can create animated objects that move about the screen by drawing, erasing, and redrawing. But these tools destroy anything they are drawn on. For instance, if the screen contains a picture of a tree, and we use DrawAt to place a picture of a man on top of it, we get a tree with a 24x24 rectangle in it and a man within the rectangle. The picture is drawn "as is" overtop of the background. What we need is a way to combine the picture of the man with the tree. We want the empty pixels about the man to act as if they were transparent.

Matting is the process of merging a picture with what is on the screen by using a special matte, or mask, which indicates the portions of the picture which should be treated as transparent. If you have used QuickDraw II, you have already seen mattes used. When you create a cursor, you create a picture of the cursor and then you make a mask to indicate where the screen pixels show through. The pen mask works in a similar way: pixels marked as white show through.

Figure 3: Merging the Background with a Picture

Background   Matte Mask   Picture   Result
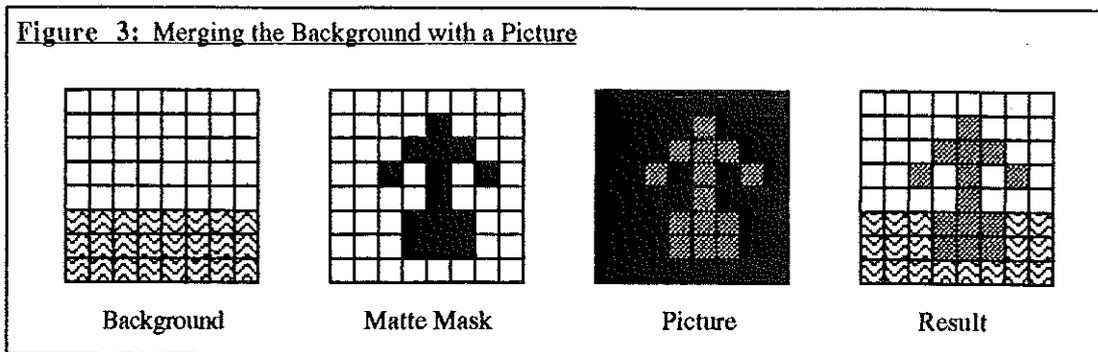
DrawTools also uses mattes to merge pictures with a background. This is done with the "on" drawing tools (DrawOn, DrawOnAt, ...). Each of these tools requires a matte to immediately follow the picture you are trying to draw. Creating a matte mask is easy. In PicEd there is a button named "mask". When you click on the button, PicEd will create a matte for the current picture and place it in the following picture position. The effect is shown in the window with the red background. When the mask is made, each black pixel in the original picture is assumed to be transparent. To view the matte, edit it. Each white pixel represents a pixel that will be taken from the background, and each black pixel represent a pixel that will be taken from the proceeding picture. It looks rather like a silhouette of the original picture.

If we want to make an entire library of matted pictures, there is even an easier way to create the matte masks. We draw pictures in each of the even numbered library positions (0, 2, 4, ..., 30). Then we can tell LoadLibrary that the masks are missing and that SetLibrary will have to generate all the masks in positions (1, 3, 5, ..., 31) for us. The following procedure shows the pictures in this kind of library. Note that the pictures will be drawn on top of whatever was previously on the screen.

Example: Drawing the contents of a picture library with matted pictures
Pascal:
```
procedure DumpOutMattedLibrary( pathname : GSOSString);
        var TheLibrary : integer;{ ID code for the library }
begin
        TheLibrary := LoadLibrary( pathname, 0, $4000);{ bit 14 = we'll need masks! }
        SetLibrary( TheLibrary );{ use this library to draw with }
        for y := 0 to 3 do{ 4 rows }
            for x := 0 to 7 do { 8 pictures per row }
                if not odd(x) then{ Skip the masks @ 1,3,... }
                    DrawOnAt( x * 32, y * 32, x + y * 8 );{ draw picture # x+8y }
end;
```

BASIC:
```
PROC DumpOutMattedLibrary
REM Load and set the library
TOOLBOX(~GenAllMasks ) : REM Generate matte masks for even-numbered pictures
FOR y% = 0 TO 3
        FOR x% = 0 TO 7 STEP 2
            Screenx% = x% * 32
```

```
        Screeny% = y% * 32
        PicNum% = x% + y% * 8
        TOOLBOX(~DrawOnAt :Screenx%, Screeny%, PicNum% )
    NEXT x%
NEXT y%
ENDPROC
```

If we don't want every black pixel to be treated as transparent, we will have to create the masks by ourselves. For instance, we may create a picture of a man with a black pixel for an eye. Create a matte mask using the mask button, and then edit the matte and remove the white pixel where the eye is. Now the eye won't be treated as transparent.

Now we can create pictures like we see in video games which move overtop of the background. However, erasing these pictures becomes a problem. We can't simply use a blank picture as we did before because the background isn't blank. To erase the pictures drawn with draw on, we need to replace the piece of the background that lay under the picture. But when we use DrawOn, we've changed the background on the screen by adding our DrawOn pictures.

The answer to this dilemma is to store the background in the shadow screen. **DrawShadow** and **DrawMain** tools let you switch whenever you want between the shadow screen and the main screen. **ShadowOff** turns off the shadow screen; you can still draw to it, but what you draw remains invisible until you "wipe" it to the main screen. **ShadowOn** turns the shadow screen on so that anything you draw will be copied to the main screen and become visible.

Figure 4: The shadow screen and animation

The main screen holds the foreground and background

The shadow screen holds an uncorrupted copy of the background
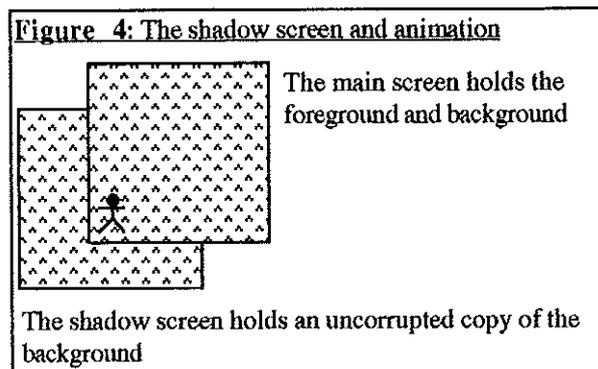
Rather than get into the details of how the shadow screen works, here's how we get around the erasing problem. First, we put the background into both the shadow screen and the main screen at the same time. The easiest way to do this is to use DrawShadow & ShadowOn and start drawing. Second, we use DrawMain and draw our matted pictures. The copy of the background that is sitting in the shadow screen remains unchanged. Finally, to erase our matted pictures, we use DrawShadow and draw empty matted pictures. Since all pixels are transparent in an empty matted picture, the background is copied to the main screen and erases any picture that we drew previously. If it sounds complicated, it is, but it's easier than trying to capture the pixels in the background each time we draw with matted pictures.

Example:This is the MoveABox procedure rewritten to move the box overtop of a background picture. It should help you put things together:

Pascal:

```
procedure MoveMattedBoxInADrawToolsLibrary( pathname : GSOSString);
const  Box = 0;{ box is picture 0 (picture 1 will be the matte mask) }
       Blank = 2;{ blank picture is picture 2 (picture 3 will be the matte mask) }
var    i, delay : integer; BoxLib : integer;
```

```
begin
        DrawShadow; { remember to specify the shadow screen in QDStartup }
        ShadowOn;
        CLS(0); { erase the shadow screen (and the main screen) }
        { Draw some stuff on the screen here - this will be in the background }
        BoxLib := LoadLibrary( pathname,   0, $4000); { load the pictures }
        SetLibrary( BoxLib ); { draw with this library }
        for i := 1 to 28 do begin{ 28 times }
            DrawMain; { switch to the main screen }
            DrawOnAt( i * 10, 20, Box); { draw a box }
            for delay := 1 to 5 do WaitVB; {time delay = 1/6 second }
            DrawShadow; { switch to the shadow scrn }
            DrawOnAt( i * 10, 20, Blank); { erase the box }
        end;
end;
```

3.8 Pixie Power: Automatic Animation

Up until now we've been looking at how to draw pictures in PicEd that we can animate and move around the screen. You could do all the animation yourself using the drawing tools to play back pictures in a specific order and erase them as appropriate. Animation involves not only pictures, but arranging them into sequences and moving the pictures about the screen. DrawTools has a special data structure to help you do just that, and it's called a pixie. It's sort of the software counterpart of a hardware sprite such have you may have seen on a Commodore 64.

A pixie is an animated object that can move around the screen. Pixies are very flexible. They can be matted or unmatted. They can have a direction or stand still. They can temporarily become invisible and then reappear somewhere else. They can use pictures from more than one library. In the DrawTools' game demo, the mother ship and the bombs it was dropping were all pixies.

Each pixie consists of two parts: 1) a sequence of picture numbers and pixie commands; 2) a data record describing the position and direction of motion. We have already seen examples of a picture sequence: we had to type in a picture sequence to do the animation examples that we did at the start of the animation section. The size of data record depends on what type of pixie you create: a simple, coarse, or fine pixie. The simple pixie is used to step through the picture sequence: it doesn't actually draw or move anything. The coarse pixie is a 24x24 bit-mapped picture that has a location and a direction. The fine pixie is similar to the coarse pixie, except that it can move with greater precision. For the rest of this section, we'll be talking about fine pixies because they are the most versatile. Most of what we'll discuss will more or less apply to the other two types.

The data structure for a fine pixie data record is already defined for you in Pascal if you are using the DrawTools interface file supplied with your DrawTools disk.

Table 5: Fine Pixie Data Record (Pascal)

Type FinePixie = record
        XVectorLow, XPositionLow : integer;
        XVectorHi, XPositionHi : integer;
        YVectorLow, YPositionLow : integer;
        YVectorHi, YPositionHi : integer;
        index : byte;
        status : byte;
end;

Here is a description of each part of the record:

XPosition - this is current position of the fine pixie (0..320, the same as the drawing tools use). If "hi" is the x-coordinate, and the "low" is in fractions of a coordinate.   Normally, you will want to leave the low's at zero.

YPosition - this is the current line number of the pixie (0...199)

XVector - this is the speed of the x direction (<0 is left, >0 is right).

YVector - this is the speed in the y direction (<0 is up, >0 is down).

Index - this is the location of the next picture in the pixie sequence; set to 0 for the first.

Status - user-defined value; we'll get to later.

BASIC: To create a fine pixie record, use the DIM statement or GET_MEM.   For example, DIM MyPixie%(9): You will have to POKE the values into the record: the offsets for the different fields are listed in the reference. You will also need to use DIM or GET_MEM to create the sequences.

For instance, XPositionHi = 100, XPositionLow = 0, YPositionHi = 50, YPositionLow = 0,   means the pixie is at (100, 50). If XVector and YVector are all zero, the pixie is standing still.

A picture sequence is simply list of bytes with the picture numbers to draw. The index to the sequence is in the data record.

Example: The following is an example of how to create a pixie of the swap disks animation that we saw in the first section.   It uses SetPixie to create a new pixie.   The constants dVisible and dFinePixie are in the DrawTools interface file and are used here just to make things easier to read. SetPixieSeq lets you select the sequence of pictures that will make up the pixie. There are also GetPixie and GetPixieSeq tools that return to you a pointer to the data record or sequence for one of the pixies.

Pascal:

```
procedure SetUpDiskSwapPixie(Dialog_Pics_Path : GSOSString);
type    APictureSequence = array[0..9] of byte;
var     Pics : APictureSeqence;
        DiskPixie : FinePixie;
        DialogLib : integer;
begin
        Pics[0] := 16; { list of pictures }
        Pics[1] := 17;Pics[2] := 18;Pics[3] := 19;{ in the animation }
        Pics[4] := 20;Pics[5] := 21;Pics[6] := 22;
        Pics[7] := 23;Pics[8] := 255;Pics[9] := 0;
        DialogLib := LoadLibrary(Dialog_Pics_Path, 0, 0);{ load the dialog pics }
        SetPixie(0, dVisible+dFinePixie, @DiskPixie);{ pixie 0 visible & a fine pixie }
        SetPixieSeq(0, DialogLib, @Pics);{ pixie uses dialog.pics }
        { & the picture sequence }
        with DiskPixie do begin
            XPositionLow := 0;XVectorLow := 0;{ place it at (50,50) }
            XPositionHi := 50;XVectorHi := 0;{ and don't move around! }
            YPositionLow := 0;YVectorLow := 0;
            YPositionHi := 50;YVectorHi := 0;
            Index := 0;{ the first picture is }
            { the first in the array}
        end;
end;
```

Once a pixie is created, it is easy to animate it with the AnimatePixie tool.   Animate pixie moves the pixie (if necessary) and then uses the drawing tools to draw the pixie. Note: it doesn't erase the pixie for you, but we don't have to erase anything for this pixie because it isn't matted nor is it moving around.

Example: How to animate a single non-matted fine pixie.
Pascal:
```
procedure AnimateDiskSwapPixie(Dialog_Pics_Path : GSOSString);
        { --- you can fill this in from the above example }
        DialogLib := LoadLibrary(Dialog_Pics_Path, 0, 0);{ load the dialog pics }
        SetPixie(0, dVisible+dFinePixie, @DiskPixie);{ use pixie #0 }
        SetPixieSeq(0, DialogLib, @Pics);{ pixie uses dialog.pics }
        { --- Initialise the data record in here }
        SetLibrary( DialogLib );
        for i := 1 to 100 do begin
            for delay := 1 to 5 do WaitVB;{time delay = 1/6 second }
            AnimatePixie( 0 );{animate pixie #0}
        end;
end;
```

3.9 Pixie Commands

A pixie sequence can contain pixie commands. A command is a special instruction for the pixie, such as to change the pixie's vector or to switch to a different library. We have already seen one pixie command: 255 is "end of sequence" command and every sequence must end with it. The number following the end of sequence command is the position in the sequence to loop back for the next picture. In our disk swapping animation, we are looping back to position 0, the start of the sequence in order that the sequence will keep repeating over and over again.

There are eight command that you can use with pixies, and they are outlined as follows:

255 - End of Sequence (All Pixies)
Marks the end of a sequence; it's followed by the index for the sequence to loop to. It can also be used to to jump forward in a sequence.

254 - Change Library (Coarse or Fine)
Switches to a different library; it's followed by a logical library number (The second parameter in LoadLibrary).

253 - Change X Vector (Fine)
Changes XVectorLow & XVectorHi; it's followed by the new low and high values. eg. 254, 0, 0, 2, 0 changes low to 0 and high to 2.

252 - Change Y Vector (Fine)
This works the same way as Change X Vector.

251 - Change X & Y Vectors (Fine)
Changes X Vector then Change Y Vector, a total of 8 new bytes plus the command byte.

250 - Change Vector (Coarse)
Changes the vector word for a coarse pixie.

249 - Change X & YVectors Relative (Fine)
This command works like 251 except that it ADDS the new vector values to the old ones.

For example, you have a sequence of an aeroplane and you want to make the aeroplane bounce during the sequence. There is no way to know the X & Y vector values, so you use 249, 0, 0, 0, 0, 0, 0, 1, 0, <picture> , 249, 0, 0, 0, 0, 0, 0, 254, 255, <picture>, 249, 0, 0, 0, 0, 0, 0, 1, 0. If the aeroplane Y vector is $0100 (dropping one line at a time) when this sequence is used, the following will happen. The first 249 causes the aeroplane Y

vector to increase by 1 so the plane drops 2 lines at a time. The second 249 changes the vector by -2 to 0. The third changes the vector back to one (the starting value). The plane does a little vertical bounce whether its gaining altitude, losing altitude, or flying straight.

<u>248 - Change Status</u> (Coarse or Fine)

The status field in the data record is for your own use. It works much like the RefCon values in things like windows. 248 is followed by the byte that you want stored in status. For instance, if you have a sequence of someone jumping, you can start the sequence with a 248, 1 and at the peak of the jump you can use a 248, 2. Now, to tell whether the player is jumping up or is starting to fall, all you have to do is check status to see if there is a 1 or 2.

<u>Example</u>: The following example is how to enbed speed changes right inside of a sequence. It's a sequence of someone jumping, where pictures 1, 2 and 3 are to be repeat during the jump. Without the speed changes, the sequence would be 1,2,3,255,0. But we want a nice looking jump where the jump starts fast (reduce the Y coordinate by 2 each animation), slows when the peak of the jump is reached (reduce by 1), and speeds up past the peak (see Figure 6). Because all these speed changes are embedded in the sequence, all our program has to do is check the pixie position to see when the jump is over.

Pascal:

```
Seq[0]  := 253;Seq[1]  := 0;Seq[2]  := 0;Seq[3]  := 254;Seq[4]  := 255;
Seq[5]  := 1;Seq[6]  := 2;Seq[7]  := 3;
Seq[8]  := 1;Seq[9]  := 2;Seq[10]  := 3;
Seq[11]  := 253;Seq[12]  := 0;Seq[13]  := 0;Seq[14]  := 255;Seq[15]  := 255;
Seq[16]  := 1;
Seq[17]  := 253;Seq[18]  := 0;Seq[19]  := 0;Seq[20]  := 0;Seq[21]  := 0;
Seq[22]  := 2;
Seq[23]  := 253;Seq[24]  := 0;Seq[25]  := 0;Seq[26]  := 1;Seq[27]  := 0;
Seq[28]  := 3;
Seq[29]  := 253;Seq[30]  := 0;Seq[31]  := 0;Seq[32]  := 2;Seq[33]  := 0;
Seq[34]  := 1;Seq[35]  := 2;Seq[36]  :=3;
Seq[37]  := 255;Seq[38]  := 34;
```


Figure 6: A Typical Jumping Sequence
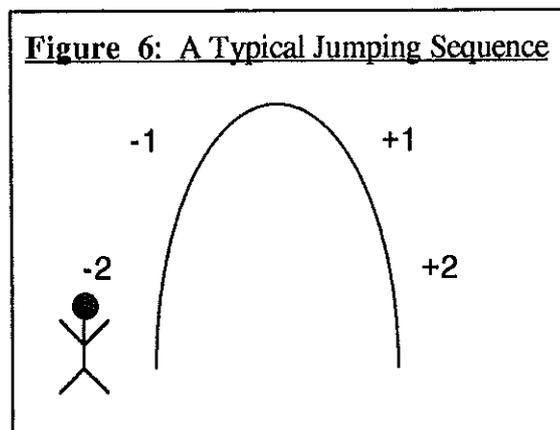
BASIC:

```
REM Assuming Seq_Addr is the address of the sequence
POKE Seq_Addr+0, 253
POKE Seq_Addr+1, 0:POKE Seq_Addr+2, 0:POKE Seq_Addr+3, 254:POKE Seq_Addr+4, 255
POKE Seq_Addr+5, 1:POKE Seq_Addr+6, 2:POKE Seq_Addr+7, 3
POKE Seq_Addr+8, 1:POKE Seq_Addr+9, 2:POKE Seq_Addr+10, 3
```

```
POKE Seq_Addr+11, 253
POKE Seq_Addr+12, 0:POKE Seq_Addr+13, 0:POKE Seq_Addr+14, 255:POKE Seq_Addr+15, 255
POKE Seq_Addr+16, 1
POKE Seq_Addr+17, 253
POKE Seq_Addr+18, 0:POKE Seq_Addr+19, 0:POKE Seq_Addr+20, 0:POKE Seq_Addr+21, 0
POKE Seq_Addr+22, 2
POKE Seq_Addr+23, 253
POKE Seq_Addr+24, 0:POKE Seq_Addr+25, 0:POKE Seq_Addr+26, 1:POKE Seq_Addr+27, 0
POKE Seq_Addr+28, 3
POKE Seq_Addr+29, 253
POKE Seq_Addr+30, 0:POKE Seq_Addr+31, 0:POKE Seq_Addr+32, 2:POKE Seq_Addr+33, 0
POKE Seq_Addr+34, 1:POKE Seq_Addr+35, 2:POKE Seq_Addr+36, 3
POKE Seq_Addr+37, 255:POKE Seq_Addr+38, 34
```

Merlin 16+:

```
        db      253 ; start jump with a new Y vector
        adrl    $FFFE0000           ; (-2) move up 2 lines for each picture displayed
        db      1, 2, 3, 1 ,2 ,3    ; display six pictures, moving up 2 lines each time
        db      253                 ; nearing top of jump; start slowing down
        adrl    $FFFF0000           ; (-1) move up one line next time
        db      1                   ; display picture, moving up one line
        db      253                 ; we're at the top of the jump; hover for one picture
        adrl    $00000000           ; don't move for next picture
        db      2                   ; picture
        db      253 ; starting to fall!
        adrl    $01000000           ; (+1) move down one line each picture
        db      3                   ; picture
        db      253                 ; fall at full speed for as long as the seq. continues
        adrl    $02000000           ; (+2) down two lines each picture
        db      1, 2, 3             ; pictures
        db      255, 34             ; end of sequence - keep repeat the last 1,2,3
```

Example: Jumping with the above pixie sequence.

Pascal:

```
DoneJumping := false;
repeat
    AnimatePixie( PixieNum );
    if HasLandedOnSomething( PixieRec.XVectorHi, PixieRec.YVectorHi ) then begin
        PixieRec.YVectorHi := 0;
        DoneJumping := true;
    end;
until DoneJumping;
```

BASIC:

```
DoneJumping! = FALSE
REPEAT
    TOOLBOX(~AnimatePixie : PixieNum% )
    XVectorHi% = PEEK(Pixie_Addr+6) + PEEK(Pixie_Addr+7) * 256
    YVectorHi% = PEEK(Pixie_Addr+14) + PEEK(Pixie_Addr+15) * 256
    IF HasLandedOnSomething[ XVectorHi%, YVectorHi% ] THEN BEGIN
        POKE Pixie_Addr+14, 0 : REM Y vector to 0
        POKE Pixie_Addr+15, 0
        DoneJumping! = TRUE
    ENDIF
```

UNTIL DoneJumping! = TRUE

3.10 Managing Multiple Pixies

We know know enough to create an animated figure that can move about the screen, even overtop of a background. The final topic here is animating multiple pixies at once, especially about how to be careful when erasing pixies.

Throughout our pixie examples we've been using pixie 0 to do our animation. DrawTools supports up to 16 pixies at once (0 ...15). You can select any one of these pixies for your animation. However, there may be occaisions when you don't care which number you use. You could make a game where new bad guys can appear at random. At any point in the game, you may not be sure of how many bad guys you have already on the screen, nor do you know which pixies are being used. To make things like this a little easier, DrawTools provides two tools called NewPixie and ClearPixie. NewPixie returns the number of the first pixie that is not being used, starting from 15 and working down towards 0. ClearPixie lets you free up a pixie that you aren't going to use anymore.

If you temporarily want to suspend a pixie without using ClearPixie to free up it's data record and sequence information, there is DisablePixie tool. When a pixie is disabled, it will not be drawn or moved, but it still exists and can be "started up" again by using EnablePixie. A pixie may also be rendered invisble by using HidePixie. A hidden pixie will move around the screen, but it won't be drawn. It appears again with a ShowPixie call. In the demo game included on the DrawTools disk, a bomb is disabled when it hits the bottom of the screen and it remains disabled until the Mother Ship is ready to drop it again. The Mother Ship is made invisible at one point in the game by using HidePixie.

Using several pixies is easy with the Animate command. It works the same way as AnimatePixie, but it animates all the enabled pixies at once, and automatically calls SetLibrary when necessary. Animate works from pixie 15 down to pixie 0. If you have two matted pixies overlapping, the pixie with the lower number will be drawn on top of the other one. Keep this in mind if the order of drawing is important. If you want a pixie airplane to fly behind a pixie cloud, the cloud must have a lower pixie number.

The most difficult aspect of working with multiple pixies is erasing. Like AnimatePixie, Animate doesn't do any erasing. This is for two very good reasons. First, Animate can't tell which picture is blank, or in what library it is in, to use for erasing. Secondly, when you are animating more than one pixie at a time and they overlap each other, the order of erasing is very important. All the pixies must be erased before they are animated since overlapping pixies will interfere with each other. Some pixies may not even need erasing, such as non-matted pixies with wide a wide border of pixels that squashes old pixels as it moves slowly arcoss the screen (as in AniDemo).

However, there are two tools to make erasing matted pixies easy. ErasePixie erases a matted pixie by copying the background on top of the pixie: this works with both coarse and fine matted pixies (that are not disabled, of course).

Example: EraseAllPixies will erase all that matted pixies. The main loop of a simple arcade game would look something like this:

Pascal:
```
Done := false;
repeat
        EraseAllPixies;
        { move the pixies }
        Animate;
until Done;
```

BASIC:
```
Done! = FALSE
REPEAT
```

```
        TOOLBOX(~EraseAllPixies )
        REM move the pixies
        TOOLBOX(~Animate )
UNTIL Done! = TRUE
```

4. Other Functions

4.1 Random Number Functions

DrawTools has three convenient random number tools. These all use QuickDraw II's Random, which returns a random integer. RND returns a random integer between 1 and the another number, like BASIC's RND function. Odds is a boolean function that is true the given percentage of the time. NormalRND is a special funtion that returns a normally distributed (bell curved) number between 1 and another number.

You can use SetRandSeed to set the "seed" for DrawTools' functions as well as Random. (The seed determines which random numbers will appear. If you set the seed to a certain the number, the random numbers returned by Random appear in the same order.).

Example: Suppose your are writing an adventure game. The player could find a treasure chest, and the chest may be booby-trapped to explode 30% of the time. If the chest doesn't explode, the player gets 10 to 15 pieces of gold. You could program it like this:

Pascal :
```
if Odds(30) then
    ExplodeChest
else
    GoldPieces := 9 + RND(6);
```

BASIC :
```
TOOLBOX(~Odds : 0, 30; Result% )
IF Result% <> 0 THEN GOSUB ExplodeChest
IF Result% = 0 THEN BEGIN
    TOOLBOX(~RND :0, 6; Result%) : REM or use BASIC's RND
    GoldPieces% = 9 + Result%
ENDIF
```

Example: A player in your game could also pick up a shovel lying abandoned in a corridor, and you want the shovel to break after an average of 20 uses. If ShovelUses is a variable with the number of good uses left in the player's shovel, you could write this:

Pascal: ShovelUses := NormalRND(40);

BASIC: TOOLBOX(98, 101: 0, 40; ShovelUses%)

With NormalRND, ShovelUses will usually have a value near 20 (half way between 1 and 40). However, there is a small chance the the shovel could have as many as 40 uses (a super-shovel) or as few as 1 use (a real "lemon").

4.2 Reading the Joystick

There are 3 tools for reading a joystick on your IIGS. To test the joystick buttons there are two tools: GetFire and StillFiring. StillFiring is the easiest to use; it is 0 if the joystick buttons are down, and greater than 0 if they are up. GetFire is only greater than 0 when a button is first down. If a button is held down, GetFire will be 0 until the button is released and pressed again. The actual number returned by these tools is a sum: button 0 has a value of 1, button 1 has a value of 2, and both buttons have a value 1 + 2 = 3.

GetJoy will determine the position of the joystick, either horizontally or vertically. GetJoy(0,0) returns the horizontal position: a value <0 if the joystick is held to the left, 0 if it's in the center, or >0 if its held to the right.

GetJoy(0,1) is <0 for up, 0 for centered, and >0 for down.

Example: Using GetJoy and StillFiring in a game.

Pascal:

```
if OnALadder then
    VerticalDir := GetJoy(0,1);
else
    HorDir := GetJoy(0,0);
if StillFiring(0) > 0 then FireGun;
```

BASIC:

```
IF OnALadder! THEN BEGIN
    TOOLBOX(~GetJoy : 0, 0, 1; VertDir%)
ELSE BEGIN
    TOOLBOX(~GetJoy : 0, 0, 0; HorDir%)
ENDIF
TOOLBOX(~StillFiring : 0, 0; Buttons%)
IF Buttons% > 0 THEN GOSUB FireGun
```

4.3 Game and Network Drivers

The newest version of DrawTools will let these 3 tools work with devices other than a joystick provided you have a game driver. A game driver in DrawTools operates a substitute device for a joystick, like the keyboard or a trackball. Up to 4 game drivers can be used at one time. (For more information on how game drivers work, consult Appendix D of the reference.)

Three sample game driver is included in the DT.Drivers folder on the DrawTools' disk:

Joystick.Drvr - simply operates the IIGS joystick using GetJoy, GetFire & StillFiring

Keypad.Drvr - simulates a joystick on the IIGS keyboard (with the Event Manager's GetNextEvent)
  • keys 1...9 specify your direction
  • 0, -, +, * are fire buttons 0, 1, 2, and 3 respectively
  • . allows you to change your speed (-2,0,+2) or (-1,0,+1)

Keypad.Drvr - simulates a joystick on the IIGS keyboard (with the Event Manager's GetNextEvent)
  • keys y,u,i,h,j,k,b,n,m specify your direction
  • space,a,s,d are fire buttons 0, 1, 2, and 3 respectively
  • f allows you to change your speed (-2,0,+2) or (-1,0,+1)

Not all languages support the system loader directly: to load a driver, you can use **LoadDriver**. To start the driver, use the DrawTools' **SetGameDriver** tool.

Example: Loading and starting a game driver (as device #1).

Pascal:

```
DriverPtr := LoadDriver( DriverPath );
SetGameDriver( 1, driverPtr );
```

BASIC:

```
REM LoadDriver requires a Pascal string.
TOOLBOX(~LoadDriver : 0, 0, PathH%, PathL%; DriverPtrL%, DriverPtrH%)
TOOLBOX(~SetGameDriver: 1, DriverPtrH%, DriverPtrL%)
```

Now whenever you use GetJoy, GetFire or StillFiring with a 1 (not 0) as the first parameter, DrawTools' will use the new device in place of a joystick.

There is also a second kind of driver you can install, a net driver, that keep you informed of devices operating on other IIGS's across a computer network or a modem.

Example: You are writing a Tetris™ clone to work with 2 players on a network. The object of the game is to be the player who survives the longest. What we need to do is:
        (1) use SendNetwork to synchronize the start of the game on two different computers
        (2) use SendNetwork to find out who "died" first.

Pascal:

```
Const   AbortGame = 8;                                {SendNetwork code to abort a game}
        GameAborted = 1;                              {S.N. code for someone aborting a game}
        ReadyToGo = 16;                               {define our own code to signal}
                                                      { that we're ready to begin playing}
        NoCmd = 0;                                    {S. N code for no command}
Var     Cmd : integer;                                { SendNetwork command word }
        Data : integer;                               { S.N. data word }
....
begin {main program}
{ do any initialization }
repeat
        Cmd := ReadyToGo;                             {signal the other IIGS that}
                                                      { we are ready }
        SendNetwork( Cmd, Data );                     { check the network }
        if Cmd <> ReadyToGo then Cmd := NoCmd;        {ignore everything unless other IIGS}
                                                      { is ready, too}
until Cmd = ReadyToGo;
{ both IIGS's will only get here by both sending "ReadyToGo" over the }
{ network.  This process is sometimes called "handshaking" }
ImDead := false;
repeat
        { do the Tetris stuff in here }
        if ImDead then                                { if player "died"/lost }
            Cmd := AbortGame;                         { inform other GS that we lost first }
        else
            Cmd := NoCmd;                             { else just check the network }
        SendNetwork( Cmd, Data );
until ImDead or (Cmd = GameAborted);                  { done if dead or other player dead }
if ImDead then
        WriteLn('You lost to the other player.')
else
        WriteLn('You won!!');
end;
```

4.4 Printing Tools

For assembly language programs, these are a simple set of tools for displaying Pascal strings and integers on the super hi-res screen. Several of the printing tools have a mode word that comes after the rest of the parameters: with this mode word, you can specify whether you want a carriage return, the rest of the line to be cleared, or if you want to tab over to a new column.

Example: Printing in assembly language.
Merlin 16+:
```
        _Ready2Print                          ; in any new grafport, use _Ready2Print
```

```
        _Home                               ; home the cursor to top of screen
        ~Print #String1;#0                  ; display Pascal string String1
        ~PrintInt #$1234;#$8000             ; display value of $1234 & do a C/R
        ~Print #String2;#$8000              ; display String2 and do a C/R
...
String1    str  'The number is '
String2    str  'All done.'
```

Output:

```
The number is 4660
All done.
```

4.5 Interrupt Tasks

In Super-Hi-Res graphics mode, the computer can be interrupted when a certain line is about to be drawn by the monitor and perform some quick task. By using interrupts, you can, for instance, have several different border colours, or can cause different sets of palettes to be available (512 colours or more instead of 256). Each task has a task header, which, strangely enough, need not be at the head of the task at all. A task could have more than one header, one for each line which is to invoke the task. Needless to say, don't touch the header if the task has been added. Once you've defined (and, hopefully, debugged before hand) your tasks, enable the interrupts (EnableSCBInts), add the tasks (SetSCBInt), and start the execution of the tasks (ResumeSCBInts).

Example: How to put 512 colours on the screen instead of only 256.
To do this, we need two sets of 16 palettes: one for the top half of the screen, and one for the bottom half. We can use the interrupt tools to switch the palettes around. Once _ResumeSCBInts is used, the palettes will be swapped "in the background", and the main program can do other things.

Merlin 16+:

```
              ~EnableSCBInts #-1
              ~SetSCBInt #Line99Header
              ~SetSCBInt #Line199Header
              _ResumeSCBInts                          ;
              ...
Line99Header  adrl 0                      ;reserved
              dw 99                       ;the line this header applies to
              dw $D44D                    ;signature word
              adrl SwapPalettes           ;invoke swap palettes on line 99
Line199Header adrl 0
              dw 199
              dw $D44D
              adrl SwapPalettes           ;invoke again at 199
SwapPalettes                              ;our interrupt task
                  phd
                  phb
* swap, in and out, the 16 palettes here
                  plb
                  pld
                  rtl
```

4.6 Scolling the Screen

The IIGS is too slow to scroll a screen quickly enough without ugly, slanting jaggies appearing as different portions of the screen are at different stages of scrolling. DrawTools has a couple of screen scrolling tools to shift the screen contents and fill the void created with a picture. To scroll faster, the screen is divided up into small blocks: if two adjacent blocks look the same, they are left alone; if they differ, they are scrolled. The result is that it appears that the whole screen is scrolling, but only the portions that need to be moved are moved.

Figure 7: A Scrolling Block
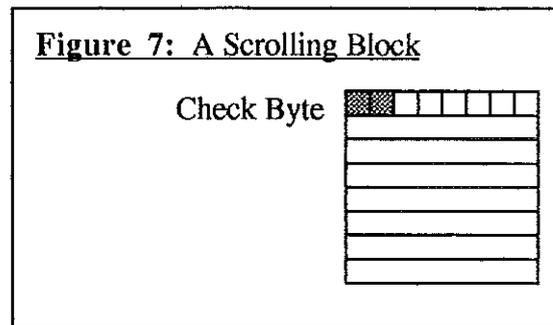
Check Byte

Each block is four bytes wide, and eight bytes high; on the screen, that's up to 40 blocks across, 25 blocks down. The upper-left byte in a block is called the check byte. If the check bytes of two adjacent blocks match, the blocks are assumed identical and no scrolling takes place. Obviously, not every picture can be scrolled using this method. Pictures must be carefully constructed, making sure the check bytes differ whenever a block differs from a neighbour. By this method, and clever art work, a picture can be made to look smooth and natural, and still scroll very quickly.

You can make two check bytes look the same but be treated as differing by using a pixel whose colour is equal to another (eg. two greens (#1,#2) of the same shade; one check byte can use green #1, and the other green #2 - they look the same, but they are actually different byte values).

The scroll tools use a scroll record, containing a description of the area of the screen to scroll, and of the picture to be scrolled in. Scrolling may extend between any two screen lines, provided that the range is composed of complete blocks (8 lines each). The scroll record parameter for width allows any rectangular picture to be scrolled onto the screen. A screen wide picture has a width of 160; DrawTools pictures have a width of 12.

4.7 Other Tools

DrawTools has a variety of other tools that may be useful in many programs.

- the work cursor, a pair of rotating gears, an alternative to the watch cursor.
- HLoad and HSave, to quickly and easily load files to handles and vice versa.
- a bar graph drawer
- a tool that let's assembly language programs call certain tools at faster speeds
- GetMHz returns the speed of the GS to the nearest MHz
- a tool to print windows or the screen on your printer

Example: How to use the work cursor.
Pascal:
```
WorkCursor2(6); {animate the work cursor every 1/10th second}
for i := 1 to 20000 do begin{StillWorking calls}
```

```
        j := j + 1;
        StillWorking;
    end;
    InitCursor;
```

BASIC:

```
    TOOLBOX(~WorkCursor2 : 6)
    FOR i% = 1 TO 20000
        j% = j% + 1
        TOOLBOX(~StillWorking )
    NEXT i%
    TOOLBOX(4, 202)
```

Merlin 16+:

```
        ~WorkCursor2 #6
        LDA #20000
        STA i
loop    INC j
        _StillWorking
        DEC i
        BNE loop
        _InitCursor
```

Example: Drawing a packed super hi-res screen (filetype $C1/$0001). This format is used by 8/16 Paint™ Screen Pictures and DreamGraphix™ PackBytes 16/256. For other programs, save the picture as an unpacked screen and use Lib.Converter 1.2 to convert the picture.

TML/Complete Pascal:

```
    P2GSString( 'MyPic', pathstr );
    PicHandle := HLoad(pathstr, $C1);
    SetBackground2( PicHandle, 0 );
```

ORCA/Pascal:

```
    PathStr.size := length('MyPic');
    PathStr.theString := 'MyPic';
    PicHandle := HLoad(pathstr, $C1);
    SetBackground2( PicHandle, 0 );
```

BASIC:

```
    REM Basic doesn't support GS/OS strings directly
    REM Use GET_MEM to get 32768 bytes, and load the picture with BLOAD.
    TOOLBOX( ~SetBackground2 : PicH%, PicL%, 0 )
```

Merlin 16+:

```
PathStr     strl 'MyPic'
. . .
        ~HLoad #PathStr; #$C1
        _SetBackground2 ; handle is still on the stack
```

II. Reference

Introduction

This is section explains the layout of the reference section, and defines some of the terms used. For a more general introduction to DrawTools, please read DrawTools Introduction manual.

Layout of Tool Entries

DrawTools provides over 100 tools. For convenience, these tools are divided up into different categories by use:

Housekeeping Tools, Low-level DrawTools, Drawing Tools, Library Management Tools, Animation Tools, Screen Tools, Scrolling Tools, Palette and Colour Tools, SCB Interrupt Tools, Printing Tools, Driver Tools and Miscellaneous Tools.

Each individual tool is described in the following format:

<u>DrawVersion ($0462)</u>   *The tool name and number.*
Returns the version number of DrawTools.   *A description of its use.*
Examples :   int := DrawVersion;   *An example in Pascal & BASIC.*
                TOOLBOX(98, 4 : 0; int%)   *(BASIC: Include 0's for each result word!)*
Parameters :   int (word) - the version, ie. $0301   *A description of each parameter.*
Errors :   none   *A description of eany errors it may return.*

Definitions of Terms

Here's an explaination of some of the terms you may encounter:
Absolute Screen Position: A pixel number, 0...31999; ASP = (x / 2) + y * 160.
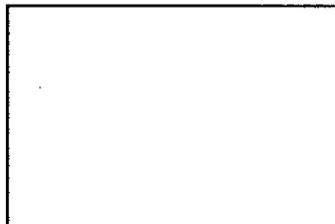
Pixel 0   Pixel 159 (159, 0)'
Pixel 160

Pixel 31999 (319, 199)

Booleans & BASIC: Treat the booleans as an integer: 0 means false, and anything else is true.
Bound lines: Bound lines are used to specify a range of screen lines. In DrawTools, bound lines need not be in ascending order.
Colour word: An RGB colour word of the form $0RGB, where R,G,B are the amounts of red, green & blue.
Current Drawing Screen: Some tools will work with either the shadow screen or main screen, whichever is active.
Library buffer: a 9K area in back 0 where recently used libraries are kept.
Main Screen: the slow drawing area in back $E1, used by most applications.
Nil pointers & BASIC: use zeroes.
Shadow Screen: the fast drawing area in bank $01.

```
┌─────────────────────────────────────────────┐
│           Housekeeping Tools                  │
│                                               │
│   These are the standard tools in every toolset. │
└─────────────────────────────────────────────┘
```

DrawBootInit   ($0162)
Should never be called by an application; does nothing.
Examples :	should never be called.
Parameters :	none
Errors :	none

DrawStartUp   ($0262)
Starts up DrawTools for use by an application. It must be made before any other DrawTools call. It does the following:

* Searches for the special QuickDraw locations
* Saves user ID with auxiliary type $F (used for all memory allocation, including HLoad's & LoadLibrary's)
* Allocates one library buffer (about 9K in bank 0)

Examples :	DrawStartUp(dpage, MMID)
	TOOLBOX(98, 2 : dpage%, MMID%)
Parameters :	dpage (word) - address of direct page workspace
	MMID (word) - memory manager ID of your application
Errors :	none

DrawShutDown   ($0362)
Shuts down DrawTools when an application quits. This routine does the following:

* Ensures the system interrupt manager is in its normal state
* Deallocates all memory used (including HLoaded handles & picture libraries)
* Shuts down the net driver, if one is installed
* Restores shadowing to its original state

Examples :	DrawShutDown;
	TOOLBOX(98, 3)
Parameters :	none
Errors :	none

DrawVersion   ($0462)
Returns the version number of DrawTools.
Examples :	int := DrawVersion;
	TOOLBOX(98, 4 : 0; int%)
Parameters :	int (word) - containing $0301, meaning version 3.1
Errors :	none

DrawReset   ($0562)

Resets DrawTools; disables SCB interrupts. This tool must not be used by an application.

Examples :   should never be called
Parameters :   none
Errors :   none

DrawStatus   ($0662)

Indicates whether DrawTools is active.

Examples :   bool := DrawStatus;
                    TOOLBOX(98, 6 : 0; bool%)
Parameters :   bool (word) - TRUE if DrawTools has been started up.
Errors :   none

> # Low-LevelTools
>
> These are tools for changing DrawTools' parameters and/or performance.

ExtendBuffers ($4A62)
Allocates as many library buffers as possible. Use this to reduce the time it takes to switch between picture libraries in graphic intensive programs (like games).

Examples :	ExtendBuffers;
	TOOLBOX(98, 74)
Parameters :	none
Errors :	$6209 - not enough bank 0 memory for another buffer
	$620A - already have maximum number allocated

ResetBuffers ($7062)
Clears the library buffers. The library buffers act as a caching mechanism for libraries: ResetBuffers clears the cache memory. Use this when you are going to using a new set of libraries. For example, when you begin a new level in a game, ResetBuffers will let you access new level libraries more efficiently.

Examples:	ResetBuffers;
	TOOLBOX(98, 112)
Parameters:	none
Errors:	none

DrawPos ($0B62)
Returns the absolute screen position for the next bit-mapped picture call.

Examples :	int := DrawPos;
	TOOLBOX(98, 11 : 0; int%)
Parameters :	int (word) - 0...31999
Errors :	none

SetDrawPos ($0C62)
Sets the absolute screen position for the next bit-mapped picture operation.

Examples :	SetDrawPos(int);
	TOOLBOX(98, 12 : int%)
Parameters :	int% (word) - 0...31999
Errors :	$62FF - the position is off the screen

DrawPage ($0D62)
Returns the location of the buffer for the current picture library.

Examples :	int% = DrawPage;
	TOOLBOX(98, 13 : 0; int%)
Parameters :	int (word) - the bank 0 location of the active set of pictures
Errors :	none

SetDrawPage   ($0E62)

Sets the location of the current picture buffer.

Examples :   SetDrawPage (locn);
                    TOOLBOX(98, 14 : locn%)
Parameters :   locn (word) - the bank 0 location of the active set of pictures
Errors :   none

DrawMain   ($0F62)

Directs DrawTools to use the main screen (bank $E1). The current grafport is also forced to the main screen.

Examples :   DrawMain;
                    TOOLBOX(98, 15)
Parameters :   none
Errors :   none

DrawShadow   ($1062)

Directs DrawTools to use the shadow screen (bank $01). The current grafport is forced to the shadow screen instead of the main screen. Micol Advanced BASIC's shell interferes with this command, but it will work in stand-alone applications.

Examples :   DrawShadow;
                    TOOLBOX(98, 16)
Parameters :   none
Errors :   none

```
                    ┌─────────────────────────────────────────────────────┐
                    │                  Drawing Tools                        │
                    │                                                       │
                    │  These tools draw bit-mapped pictures from picture libraries, or produce masks │
                    │  for matted bit-mapped pictures, without any animation. If you are using pixies, │
                    │                 see the animation tools.              │
                    └─────────────────────────────────────────────────────┘
```

Draw ($0962)
Draws a 24x24 bit-mapped picture at the current screen position and advance to the right.
Examples :	Draw(pic);
	TOOLBOX(98, 9 : pic%)
Parameters :	pic (word) - picture in the current library (0...31)
Errors :	none

Draw48 ($0A62)
Draws a 48x48 bit-mapped picture at the current screen position and advances to the right.
Examples :	Draw48(pic);
	TOOLBOX(98, 10 : pic%)
Parameters :	pic (word) - picture in the current library (0..28)
Errors :	none

DrawAt ($1462)
Draws a 24x24 bit-mapped picture at the specified screen position and advances to the right.
Examples :	DrawAt(xco, yco, pic);
	TOOLBOX(98, 20 : xco%, yco%, pic%)
Parameters :	xco (word) - x-coordinate (0..319)
	yco (word) - y-coordinate (0...199)
	pic (word) - picture in the current library (0..31)
Errors :	none (if bad coordinates are used, the picture is drawn at the upper-left corner)

Draw48At ($1562)
Draws a 48x48 bit-mapped picture at the specified screen position and advances to the right.
Examples :	Draw48At(xco, yco, pic);
	TOOLBOX(98, 21 : xco%, yco%, pic%)
Parameters :	xco (word) - x-coordinate (0..319)
	yco (word) - y-coordinate (0...199)
	pic (word) - picture in the current library (0..28)
Errors :	none (if bad coordinates are used, the picture is drawn at the upper-left corner)

DrawOn ($2262)
Draws a matted 24x24 bit-mapped picture at the current screen position and advances one position to the right.
Examples :	DrawOn(pic);
	TOOLBOX(98, 34 : pic%)
Parameters :	pic (word) - picture in the current library (0..30)
Errors :	none

Draw48On   ($2362)

Draws a matted 48x48 bit-mapped picture at the current screen position and advances to the right.

Examples :   Draw48On(pic);
                    TOOLBOX(98, 35 : pic%)
Parameters :   pic (word) - picture in the current library (0..24)
Errors :   none

DrawOnAt   ($2462)

Draws a matted 24x24 bit-mapped picture at the specified screen position and advances to the right.

Examples :   DrawOnAt(xco, yco, pic);
                    TOOLBOX(98, 36 : xco%, yco%, pic%)
Parameters :   xco (word) - x-coordinate (0..319)
                    yco (word) - y-coordinate (0...199)
                    pic (word) - picture in the current library (0..30)
Errors :   none (if bad coordinates are used, the picture is drawn at the upper-left corner)

Draw48OnAt   ($2562)

Draws a matted 48x48 bit-mapped picture at the specified screen position and advances to the right.

Examples :   Draw48OnAt(xco, yco, pic);
                    TOOLBOX(48, 37 : xco%, yco%, pic%)
Parameters :   xco (word) - x-coordinate (0..319)
                    yco (word) - y-coordinate (0...199)
                    pic (word) - picture in the current library (0..24)
Errors :   none (if bad coordinates are used, the picture is drawn at the upper-left corner)

GenMask   ($2162)

Generates a matting mask for the specified picture and stores it in the next picture position.

Examples :   GenMask(pic);
                    TOOLBOX(48, 33 : pic%)
Parameters :   pic (word) - picture in the current library to make a mask for (0..30)
Errors :   none

GetAllMasks   ($2662)

Generates a matting mask for every even-numbered picture in the current picture buffer, storing each mask in the following odd-numbered picture position.

Examples :   GenAllMasks;
                    TOOLBOX(48, 38)
Parameters :   none
Errors :   none

SetBackground† ($3962)

Draws a packed super hi-res screen (filetype PNT/$0000) on the current drawing screen. The handle is left unlocked.

Examples :   SetBackground(Background);
                    TOOLBOX(98, 57 : BackgroundH%, BackgroundL%)
Parameters :   BackgroundHandle (long) - handle to packed picture
Errors :   memory manager errors

† See SetBackround2.

WipeOn† ($5562)

Wipes a 24x24 block of pixels from the shadow screen to the main screen at the current drawing position. Shadowing must be enabled.

Examples :   WipeOn;
                    TOOLBOX(98, 85)
Parameters :   none
Errors :   none

† For use with pixies, see ErasePixie and EraseAllPixies.

SetBackground2 ($6F62)

Draws a packed super hi-res screen (filetype PNT/$0000) on the current drawing screen. You can create this kind of picture by packing a super hi-res screen with PackBytes, or using one of several IIGS graphics conversion utilities that are available, or by saving an 8/16-Paint™ picture as a screen. The handle is left unlocked.

Examples: SetBackground2(Background, Flags);
                    TOOLBOX(98, 111 : BackgroundH%, BackgroundL%, Flags%)
Parameters: Background (long) - handle to the packed screen
                    Flags (word) - list of options:
                        0 - normal (like SetBackground)
                        1 - pixels and SCBs only (no palettes) for QuickWipe or VBWipe
                        2 - ready to fade in with QuickFadeIn or IncrFadeIn
Errors: memory manager errors

```
┌─────────────────────────────────────────────────┐
│            Library Management Tools             │
│                                                 │
│      Tools used in loading and using picture libraries.      │
└─────────────────────────────────────────────────┘
```

LoadLibrary ($2B62)

Retrieves a DrawTools picture library from the disk and returns its library ID number.

Examples :   LibID := LoadLibrary(path, SeqLibNum, packed);
      TOOLBOX(48, 43 : 0, PathH%, PathL%, SeqLibNum%, Packed% ; LibID%)

Parameters : path (long) - GS/OS path name pointer
      SeqLibNum (word) - logical number for pixie sequence commands (else just 0)
      Packed (word) - bit 15 - TRUE if the library is packed with PackBytes
          - bit 14 - TRUE if GenAllMasks should be called before library is used
      LibID (word) - the ID number for the library

Errors : $6201 - too many libraries loaded (current limit is 24)
      $6202 - SeqLibNum is out of range
      GS/OS and Memory Manager errors

UnloadLibrary ($2E62)

Deallocates a library loaded with LoadLibrary. Normally, DrawShutDown automatically unloads all libraries. However, this tool allows you to manually discard a library you no longer need without shutting down DrawTools.

Examples :   UnloadLibrary (LibID);
      TOOLBOX(48, 46 : LibID%)

Parameters : LibID (word) - the ID of the library to unload

Errors : $6203 - invalid library ID number
      $6204 - the library isn't loaded
      Memory Manager errors

SetLibrary ($2C62)

Makes the specified library the current one use with the drawing or animation tools.

Examples :   SetLibrary(LibID);
      TOOLBOX(48, 44 : LibID%)

Parameters : LibID (word) - the ID of the library to make current

Errors : $6203 - invalid library ID number
      $6204 - the library isn't loaded
      Memory Manager errors

GetLibrary ($2D62)

Returns the library id of the current library.

Examples :   LibID := GetLibrary;
      TOOLBOX(48, 45 : 0; LibID%)

Parameters : LibID (word) - the ID of the current library (-1 if none)

Errors :   none

<div style="border:1px solid">

Animation Tools

Tools used in animating objects & handling animation sequences.

</div>

Fine Pixie Data Record:
0,1 - X Vector Low (word)
2,3 - X Position Low (word)
4,5 - X Vector Hi (word)
6,7 - X Position Hi (word)
8,9 - Y Vector Low (word)
10,11 - Y Position Low (word)
12,13 - Y Vector Hi (word)
14,15 - Y Position Hi (word)
16 - Index (byte)
17 - Status (byte)

Coarse Pixie Data Record:
0,1 - Vector (word)
2,3 - Position (word)
4 - Index (byte)
5 - status (byte)

Simple Pixie Data Record:
0 - Index (byte)
1 - Last Frame (byte)

When you animate a pixie, the new pixie position is calculated by adding the vector value to the position value, resulting in the new position. For example, a fine pixie with an x vector of 1 (hi 1, low 0) and an original x position of 10 (hi 10, lo 0) will move to x position 11 the next time it is animated.

Sequence Byte	Description	Parameter Bytes Following Byte
0...31	picture to use in current library	-
32...247	reserved	-
248	change status byte	new status (byte)
249	change fine pixie dir relative	X & Y vectors to add to current vectors (4 words)
250	change coarse pixie direction	new direction (word)
251	change fine pixie direction	new X & Y words (4 words)
252	change fine pixie y direction	new Y words (2 words)
253	change fine pixie x direction	new X words (2 words)
254	change library	LoadLibrary logical number (word)
255	end of sequence	position to resume at (byte)

For simple pixies, any negative byte (128 or bigger) is considered an end of sequence command.

NewPixie ($3A62)

Returns a number of a pixie that's not in use. When allocating several pixies, remember to use SetPixie after each NewPixie, or NewPixie will return the same number each time. -1 is returned if no pixie is free.

Examples :   MyPixieNum := NewPixie;
               TOOLBOX(98, 58 : 0; MyPixieNum%)
Parameters :   MyPixieNum (word) - the pixie number (0...15)
Errors :   none

ClearPixie ($3B62)
Deallocates the specified pixie table position.

Examples :   ClearPixie(MyPixieNum);
                    TOOLBOX(98, 59 : MyPixieNum%)
Parameters :   MyPixieNum (word) - the pixie number (0...15)
Errors :   · $62FF - the pixie number is out of range, or the position is already free

SetPixie ($4E62)
Sets up a pixie for use. If that pixie already exists, the old pixie is overwritten.

Examples :   SetPixie(pixnum, pixiedesc, pixieptr);
                    TOOLBOX(48, 78 : pixnum%, pixiedesc%, pixieptrh%, pixieptrl%)
Parameters :   pixnum (word) - the pixie number (0..15)
                    pixiedesc (word) - description of the pixie:
                        bit 15 - pixie visible (TRUE) or invisible (FALSE)
                        bit 14 - pixie matted (TRUE) or not matted (FALSE)
                        bit 3-13 - reserved, set to 0
                        bit 0-2 - pixie type (0=simple, 1=coarse, 2=fine)
                    pixieptr (long) - pointer to the pixie data record
Errors : $62FF - pixie number is out of range

GetPixie ($4F62)
Returns a pointer to the specified pixie's data record.

Examples :   PixiePtr := GetPixie(PixieNum);
                    TOOLBOX(98, 79 : 0, 0, PixieNum%; PixiePtrL%, PixiePtrH%)
Parameters :   PixieNum (word) - the pixie number
                    PixiePtr (long) - pointer to the pixie data record
Errors : $62FF - pixie number is out of range

SetPixieSeq ($2A62)
Assigns the specified animation sequence to a pixie; any old sequence is overwritten. The sequence index (in the pixie data record) is not changed.

Examples :   SetPixieSeq(PixieNum, LibID%, SeqPtr);
                    TOOLBOX(98, 42 : PixieNum%, LibID%, SeqPtrH%, SeqPtrL%)
Parameters :   PixieNum (word) - the pixie number
                    LibID (word) - the default picture library
                    SeqPtr (long) - pointer to the animation sequence
Errors : $62FF - pixie number is out of range

GetPixieSeq($5062)
Returns the pointer to a pixie's animation sequence.

Examples :   SeqPtr := GetPixieSeq(PixieNum);
                    TOOLBOX(98, 80 : 0, 0, PixieNum% ; SeqPtrL%, SeqPtrH%)
Parameters :   PixieNum (word) - the pixie number
                    SeqPtr (long) - pointer to the animation sequence
Errors : $62FF - pixie number is out of range

HidePixie   ($5262)

Stop drawing a pixie on subsequent animation calls, but continue animating it as if it were visible.

Examples :   HidePixie(PixieNum);
                    TOOLBOX(98, 82 : PixieNum%)
Parameters :   PixieNum (word) - the pixie number
Errors : $62FF - the pixie number is out of range

ShowPixie   ($5162)

Draw a pixie on subsequent animation calls.

Examples :   ShowPixie(PixieNum);
                    TOOLBOX(98, 81 : PixieNum%)
Parameters :   PixieNum (word) - the pixie number
Errors : $62FF - the pixie number is out of range

DisablePixie   ($2862)

Stop animating a pixie on subsequent animation calls.

Examples :   DisablePixie(PixieNum);
                    TOOLBOX(98, 40 : PixieNum%)
Parameters :   PixieNum (word) - the pixie number
Errors : $62FF - the pixie number is out of range

EnablePixie   ($2962)

Animate a pixie on subsequent animation calls.

Examples :   EnablePixie(PixieNum);
                    TOOLBOX(98, 41 : PixieNum%)
Parameters :   PixieNum (word) - the pixie number
Errors : $62FF - the pixie number is out of range

AnimatePixie   ($5362)

Animate a single pixie one picture along its sequence. Unlike Animate, you will have to use SetLibrary to select the picture library for the pixie. The drawing position for the drawing tools is unaffected.

Examples :   AnimatePixie(PixieNum);
                    TOOLBOX(98, 83 : PixieNum%)
Parameters :   PixieNum (word) - the pixie number
Errors : $620C - Command for a different kind of pixie (disables pixie)
                    $620D - Undefined command in sequence (disables pixie)
                    $620E - Pixie doesn't exist
                    $62FF - Pixie number is out of range
                    SetLibrary errors

Animate   ($2762)

Animates all of the pixies one picture along their sequences. The drawing position for the drawing tools is unaffected.

Examples :   Animate;
                    TOOLBOX(48, 39)
Parameters :   none

Errors : $620C - Command for a different kind of pixie (disables pixie)
                       $620D - Undefined command in sequence (disables pixie)
                       SetLibrary errors

ErasePixie   ($6B62)
Erases the specified matted pixie with the corresponding contents of the shadow screen.
Examples: ErasePixie(Pixie);
                       TOOLBOX(98, 107: Pixie%)
Parameters: Pixie (word) - the pixie number
Errors: $620E - Pixie doesn't exist
                       $6211 - Not a matted pixie
                       $62FF - Pixie number is out of range

EraseAllPixies   ($6C62)
Erases all enabled, matted pixies.
Examples: EraseAllPixies;
                       TOOLBOX(98, 108)
Parameters: none
Errors: none

```
┌─────────────────────────────────────────────────────────────────┐
│                        Screen Tools                             │
│                                                                 │
│ Tools involving the screen, including those involving shadowing and the SCBs. │
└─────────────────────────────────────────────────────────────────┘
```

CLS† ($3462)

This tool acts the same as QuickDraw II''s ClearScreen.

Examples :	CLS (ColourWord);
	TOOLBOX(48, 52 : ColourWord%)
Parameters :	ColourWord (word) - word to fill the screen with
Errors :	none

† Before System 6.0, ClearScreen would not clear the shadow screen; CLS works fine on older systems.

QuickWipe ($1C62)

This tool copies the shadow screen to the main screen.

Examples :	QuickWipe;
	TOOLBOX(48, 28)
Parameters :	none
Errors :	none

VBWipe ($3562)

This tool copies the shadow screen to the main screen using a "Venetian blind" effect.

Examples :	VBWipe;
	TOOLBOX(48, 53)
Parameters :	none
Errors :	none

FadeDone ($4C62)

Returns TRUE if a fade is finished fading. In the current version of DrawTools, all fading occurs during the In/Out call; future versions will fade during the FadeDone calls to allow animation to continue during the fading process. For compatibility, always have a REPEAT...UNTIL FadeDone (or the equivalent in your language) immediately after using a fade tool.

Examples :	done := FadeDone;
	TOOLBOX(98, 76 : 0; done%)
Parameters :	done (word) - TRUE if the last fade has been completed
Errors :	none

QuickFadeOut/In ($16/1762)

Fades the colours in the first eight palettes to black, or restores them to their original values. The upper eight palettes are used to store the original palettes.

Examples :	QuickFadeIn(rate);
	TOOLBOX(48, 22 : rate%)
Parameters :	rate (word) - # 60th's of a second between INCs/DECs
Errors :	none

IncrFadeOut/In ($18/1962)

Fades the colours in the first eight palettes to red, then to black, or restores them to their original values ("incremental fade"). The upper eight palettes are used to store the original palettes.

Examples :	IncrFadeIn(rate);
	TOOLBOX(48, 24)
Parameters :	rate (word) - # 60th's of a second between INCs/DECs
Errors :	none

ShadowOn ($1262)

This tool enables the hardware shadowing of the shadow screen. If you open a new grafport (using OpenPort) with shadowing enabled, the port will be assigned to the shadow screen.

Examples :	ShadowOn;
	TOOLBOX(48, 18)
Parameters :	none
Errors :	none

ShadowOff ($1162)

This tool disables the hardware shadowing of the shadow screen. If you open a new grafport (using OpenPort) with shadowing disabled, the port will be assigned to the main screen.

Examples :	ShadowOff;
	TOOLBOX(48, 17)
Parameters :	none
Errors :	none

WaitVB ($1362)

This tool passes time until the beginning of the next vertical blanking period (1/60 to 1/30 of a second). If you erase during a vertical planking period, you will have less flicker in your animation.

Examples :	WaitVB;
	TOOLBOX(48, 19) ⟩
Parameters :	none
Errors :	none

WaitLine ($5E62)

This tool waits until your monitor is drawing a particular line. Use this to reduce flicker when you are drawing by waiting until an object is drawn on the monitor before erasing it.

Example :	WaitLine(line);
	TOOLBOX(98, 94 : line%)
Parameters :	line (word) - line number to wait for, 0..199
	if line < 0, line is treated as 0
	if line > 199, line is treated as 200 (same as WaitVB)
Errors:	none

SetBorder ($1F62)
This tool sets the colour of the screen border.
Examples :	SetBorder(Colour);
	TOOLBOX(48, 31 : Colour%)
Parameters :	Colour (word) - the new colour (0...15) as in the control panel
Errors :	none.

GetBorder ($1E62)
This tool returns the current colour of the screen border.
Examples :	Colour := GetBorder;
	TOOLBOX(48, 30 : 0; Colour%)
Parameters :	Colour (word) - the colour (0...15) as in the control panel.
Errors :	none

SetSCBs ($3662)
This tool sets specific bits in the SCB's for a range of lines. This tool should not be used to change the interrupt bit while the SCB interrupt handler is enabled.
Examples :	SetSCBs(line1, line2, BitsToSet);
	TOOLBOX(98, 54 : line1%, line2%, Bits%)
Parameters :	line1 (word) - first bound line
	line2 (word) - last bound line
	BitsToSet (word) - mask of bits to set (1=set bit)
Errors :	none

ResetSCBs ($3762)
This tool resets specific bits in the SCB's for a range of lines. This tool should not be used to change the interrupt bit while the SCB interrupt handler is enabled.
Examples :	ResetSCBs(line1, line2, BitsToReset);
	TOOLBOX(48, 55 : line1%, line2%, Bits%)
Parameters :	line1 (word) - first bound line
	line2 (word) - last bound line
	BitsToReset (word) - mask of bits to reset (1=reset bit)
Errors :	none

```
┌─────────────────────────────────────┐
│                                     │
│          Scrolling Tools            │
│                                     │
│   Tools to scroll portions of the screen.│
│                                     │
└─────────────────────────────────────┘
```

The Format of a Scroll Record:

0,1	offset (word)	byte offset into fill picture
2,3	width (word)	width of the picture in bytes (eg. 160 for a screen image, 12 for DT pic)
4-7	fillpic (long)	ptr to picture to fill with
8,9	first (word)	first (top) screen line to scroll
10,11	numblocks (word)	number of 8 line blocks to scroll
12-15	reserved	must be 0

ScrollLinesL   ($3062)

This tool scrolls the indicated lines one block (2 words) to the left, and fills them from a specified picture. The offset is incremented by the width.

Examples :   ScrollLinesL(ScrollRec);
                    TOOLBOX(48, 48 : ScrollRecH%, ScrollRecL%)
Parameters :   ScrollRec (long) - pointer to the scroll record
Errors : $6206 - first line is out of range

ScrollLinesR   ($3162)

This tool scrolls the indicated lines one block (2 words) to the right, and fills them with a specified picture. The offset is decremented by the width.

Examples :   ScrollLinesR(ScrollRec);
                    TOOLBOX(48, 49 : ScrollRecH%, ScrollRecL%)
Parameters :   ScrollRec (long) - pointer to the scroll record
Errors : $6206 - first line is out of range

Note: The current version of ScrollLinesR will not work properly if the first line is '0.

ScrollLinesU   ($3262) (not available yet)

This tool scrolls the indicated lines one block (2 words) to up, and fills them with a specified picture. The offset is incremented by a row of blocks

ScrollLinesD   ($3362) (not available yet)

This tool scrolls the indicated lines one block (2 words) to down, and fills them with a specified picture. The offset is decremented by a row of blocks.

```
┌──────────────────────────────────────────────────┐
│                                                    │
│           Palette and Colour Tools                 │
│                                                    │
│   Tools that change colours and manipulate palettes.│
│                                                    │
└──────────────────────────────────────────────────┘
```

SetPalette ($1A62)

This tool sets the palette for a range of screen lines.

Examples :	SetPalette(line1, line2, palette);
	TOOLBOX(48, 26 : line1%, line2%, palette%)
Parameters :	line1 (word) - first bound line
	line2 (word) - last bound line
	palette (word) - new palette number for lines (0..15)
Errors :	none (no range checking)

GetPalette ($3862)

This tool returns the palette assigned to a particular screen line.

Examples :	palette := GetPalette(line);
	TOOLBOX(48, 56 : 0, line% ; palette%)
Parameters :	line (word) - which line to check
	palette (word) - palette number for that line (0...15)
Errors :	none

FadePal ($1B62)

This tool dims the source palette colours and stores them in the target palette.

Examples :	FadePal(sourcepal, targetpal);
	TOOLBOX(48, 27 : sourcepal%, targetpal%)
Parameters :s	ourcepal (word) - palette to fade (0...15)
	targetpal (word) - where to store the faded palette (0...15)
Errors :	none

UnfadePal ($1D62)

This tool brightens the source palette colours towards those in the target palette. The colours are stored in the source palette.

Examples :	UnfadePal(SourcePal, TargetPal);
	TOOLBOX(48, 29 : sourcepal%, targetpal%)
Parameters :	sourcepal (word) - palette to brighten (0...15)
	targetpal (word) - palette to compare with (0..15)
Errors :	none

SetColour   ($4062)

Combines the red, green and blue values into a colour word.

Examples :   word := SetColour(red, green, blue);
                    TOOLBOX(48, 64 : 0, red%, green%, blue%; word%)
Parameters :   word (word) - palette colour word
                    red (word) - amount of red (0...15)
                    green (word) - amount of green (0...15)
                    blue (word) - amount of blue (0...15)
Errors :   none. Bad values result in a meaningless colour word.

SetColPercent   ($4162)

Combines the red, green and blue percentage values into a colour word.

Examples :   word := SetColPercent(red, green, blue);
                    TOOLBOX(48, 65 : 0, red%, green%, blue%; word%)
Parameters :   word (word) - palette colour word
                    red (word) - percentage of red (0...100)
                    green (word) - percentage of green (0...100)
                    blue (word) - percentage of blue (0...100)
Errors :   none. Bad values result in a meaningless colour word.

Elaboration: A few example RGB percent values (extracted from ACM SIGGRAPH '89 course notes):

Gold	78, 61, 16	Old (dark) gold	78, 43, 10
Platinum	83, 79, 56	Silver	81, 82, 70
Antique (dark) silver	53, 52, 47	Steel	55, 62, 59
Copper	97, 60, 28	Brass	69, 63, 23
Iron	18, 7, 6	Sunlight	100, 96, 92
Moonligh	75, 81, 100	Naples Yellow	100,66,7
Cadmium Red (Ruby)	89, 9, 5	Brown Madder	86, 16, 16
King's Blue	1, 57, 76	Indigo	3, 18, 33
Emerald Green	0, 79, 34	Terre-verte	22, 37, 6
Ivory Grey	16, 14, 13	Lamp Black	18, 28, 23

FindColour   ($4262)

This tool search the specified palette for the closest colour to the one requested.

Example :   colour := FindColour(numcol, palette, colourWord);
                    TOOLBOX(48, 66 : 0, numcol%, palette%, colourWord%; colourWord%)
Parameters :   numcol (word) - 16 if 320 mode, 4 if 640
                    colour (word) - the colour number of the closest colour
                    palette (word) - the palette to search
                    colourWord (word) - the palette colour word to match
Errors :   none

BlendColour   ($5F62)

Blends two colours together to form a new colour.

Example :   colour := BlendColour(weight, col1, col2);
                    TOOLBOX(98, 95: 0, weight%, col1%, col2%; colour%)
Parameters :   colour (word) - the new colour word
                    weight (word) - 0..16, amount of second colour to mix in
                    col1 (word) - the first colour word
                    col2 (word) - the second colour word
Errors : $62FF - weight is out of range

Elaboration: Some BlendColour Applications:
(1) Blending: colour := BlendColour(weight, col1, col2);
(2) Bleaching (eg. for distance): colour := BlendColour(distance, col, backgroundcol);
(3) Anti-aliasing: (a) colour := BlendColour(amount in pixel, colour, backgroundcol); (b) colourNum :=
FindColour(16, 0, colour); {for 320 mode}
(4) Saturating: colour := BlendColour(how much to saturate, colour, $0F00);

FadeColour   ($6062)

Fades or brightens a colour.

Example :   colour := FadeColour(oldcolour, difference);
                    TOOLBOX(98, 96: 0, oldcolour%, difference%; colour%)
Parameters :   colour (word) - the new colour word
                    oldcolour (word) - the original colour word
                    difference (word) - (-15) to (+15), amount to change the colour by
Errors :   None

Elaboration: Some FadeColour Applications:
(1) Darken colour: colour := FadeColour(oldcolour,-1);
(2) Brighten colour: colour := FadeColour(oldcolour, +1);

FindPalette   ($6162)

This is my "mini Palette Manager" tool. Returns the colour numbers for the entries in the current palette which
most closely resemble the colours that you expect in that palette. Especially useful for NDAs, where you don't know
what colours will be on the screen. FindPalette only recognises pure colours in 640 mode (not dithered colours).

Example :   changed := FindPalette(colours, palette);
                    TOOLBOX(98, 97: 0, coloursH%, coloursL%, paletteH%, paletteL%; changed%)
Parameters :   changed (boolean) - True if the colours have changed since last FindPalette
                    colours (long) - address of a list of 16 colour numbers corresponding to the colours in the palette
                    palette (long) - address of palette (a QuickDraw II colorTable) of desired colours
Errors :   none

```
┌─────────────────────────────────────────────────────────────────┐
│                                                                   │
│                        Interrupt Tools                            │
│                                                                   │
│      Tools Involving SCB (or Horizontal Retrace, or Scan Line) Interrupts │
│                                                                   │
└─────────────────────────────────────────────────────────────────┘
```

Format of a SCB interrupt task header:

0-3	longword	TaskPtr	Use by the Interrupt Tools; do not modify
4-5	word	Scan	Line Line number of the task
6-7	word	SigWord	signature word; always $D44D
8-A	3 bytes	EntryPt	task entry pointer

Designing an Interrupt Task: The task must be a long subroutine (that is, end in an RTL instruction). B and D registers must be preserved, but other registers (A,X,Y,P) need not be. A task may have two or more headers if it is to be used on two different screen lines. Because DrawTools is non-reentrant, never call a DrawTools from a task unless your are sure the main program is not using DrawTools at the same time.

IMPORTANT: (1) I have no idea why, but if you use the SCB interrupts, make sure you unload DrawTools before your program quits or the next program that runs will crash; at least, it happens with Merlin 16+ and EXE files -> it crashes during a Misc. Tools _GetVector call in DrawStartup. (2) When the interrupts are enabled with EnableSCBInts, do not switch the processor into emulation mode (e=1) without suspending interrupts (with SEI). The patch I placed on the interrupt manager is not designed to handle emulation mode IRQs.

EnableSCBInts ($4A62)

This tool must be used before all other SCB interrupt tools. Patches the system interrupt manager to use my SCB interrupt handler.
QuickDraw SCB interrupt use is suspended. The task list is cleared.

Examples :	EnableSCBInts(enable);
	TOOLBOX(98, 74 : enable%)
Parameters :	enable (boolean) - TRUE if interrupts are to be enabled
Errors :	none

SetSCBInt ($3C62)

Installs a SCB interrupt task for the given screen line. Automatically suspends all tasks until the next ResumeSCBInts.

Examples :	SetSCBInt(TaskPtr);
	TOOLBOX(98, 58: MachineLgH%, MachineLgL%)
Parameters :	TaskPtr (longword) - pointer to the task header
Errors :	$6205 - Task signature isn't $D44D
	$6206 - The screen line is out of range
	$6207 - A task already exists for that line

DelSCBInt ($3D62)

Deletes a SCB interrupt task. Automatically suspends all tasks.

Examples :	DelSCBInt(TaskLine);
	TOOLBOX(98, 59 : TaskLine%)
Parameters :	TaskLine (integer) - screen line of the task
Errors :	$6206 - The screen line is out of range

$6207 - A task doesn't exist for that line

ClrSCBInts ($3E62)
Deletes all SCB interrupt tasks. Automatically suspends all tasks.

Example :	ClrSCBInts;
	TOOLBOX(48, 62)
Parameters :	none
Errors :	none

ResumeSCBInts ($3F62)
Waits for the next vertical blanking period and resumes executing all SCB interrupt tasks.

Example :	ResumeSCBInts;
	TOOLBOX(48, 63)
Parameters :	none
Errors :	$6208 - SCB interrupts not enabled
	$62FF - no tasks to execute

```
┌─────────────────────────────────────────────────────────────────┐
│                        Printing Tools                            │
│                                                                   │
│   Tools to help assembly language programs write on the screen.  │
└─────────────────────────────────────────────────────────────────┘
```

Ready2Print ($5662)

This must be the first printing call in a new window (or grafport). Gets a pointer to the current grafport, resets the margins to 0, and "homes" the QuickDraw pen.

Examples :   ~Ready2Print
Parameters :   None
Errors :   None

SetLTMargins ($5D62)

Sets the left and top printing margins. Use Home to place the pen in the top-left corner of the new margin settings.

Examples :   ~SetLTMargins #Left; #Top
Parameters :   Left (word) - left margin, in pixels
                   Top (word) - top margin, in pixels
Errors :   None

Home ($5762)

Moves the QuickDraw pen to the left end of the first text line on the screen, like BASIC's HOME.

Examples :   ~Home
Parameters :   None
Errors :   None

HTab ($5862)

Moves the pen the specified number of pixels to the right of the left margin.

Examples :   ~HTab #Indent   ⟩
Parameters :   Indent (word) - number of pixels to indent
Errors :   None

VTab ($5962)

Moves the pen down the specified number of screen lines from the top margin, based on the height of the current font.

Examples :   ~VTab #NewLine
Parameters :   NewLine (word) - new screen line; 1 is the top line.
Errors : $62FF - NewLine was negative or zero

Print ($5A62)

Draws a Pascal string on the screen.

Examples :   ~Print #str; #mode
Parameters :   str (long) - pointer to the Pascal string
                    mode (word) - printing mode:
                    .   bit 15 - TRUE if a carriage return is to follow printing
                                bit 7 - tab to next column of 64 pixels after printing
                                bit 6 - clear to end of the line
                                other bits - reserved; set to 0
Errors :   none

PrintHex   ($5B62)

Draws a hexadecimal value on the screen.

Examples :   ~PrintHex #number; #mode
Parameters :   number (word) - the number to print
                    mode (word) - same as with Print
Errors :   none

PrintInt   ($5C62)

Draws a signed integer value on the screen.

Examples :   ~PrintInt #number; #mode
Parameters: number (word) - the number to print
                    mode (word) - same as with Print
Errors :   none

Driver Tools

For a general discussion on game and network drivers, including how to design them, see Appendix D.

LoadDriver ($6D62)
Loads a specified game or net driver into memory.

Example:	DriverPtr := LoadDriver(DriverPath);
	TOOLBOX(98, 109: 0, 0, DriverPathH%, DriverPathL%; DriverPtrL%, DriverPtrH%)
Parameters:	DriverPath (long) - the Pascal string pathname
	DriverPtr (long) - pointer to the driver
Errors:	GS/OS errors

UnloadDriver ($6E62)
Unloads a specified game or net driver from memory.

Example:	UnloadDriver(DriverPtr);
	TOOLBOX(98, 110: DriverPtrH%, DriverPtrL%)
Parameters:	DriverPath (long) - pointer to the driver to unload
Errors:	$62FF - unknown error while unloading

SetGameDriver ($6362)
Installs a game driver for the specified player.

Example :	SetGameDriver(playerNum, driverPtr);
	TOOLBOX(98, 99: playerNum%, driverH%, driverL%)
Parameters :	playerNum (word) - 1..4, the player to use the game driver
	driverPtr (long) - address of the game driver
Errors :	$62FF - DrawTools version is too low for this driver
	$620F - device number is out of range
	$6210 - The device this driver operates can't be found on the GS

SetNetDriver ($6262)
Installs a network driver so that remote game drivers can be supported.

Example :	SetNetDriver(driverPtr);
	TOOLBOX(98, 98: driverH%, driverL%)
Parameters :	driverPtr (long) - address of the net driver
Errors :	$62FF - DrawTools version is too low for this driver
	$6210 - The device this driver operates can't be found on the GS

SendNetwork ($6462)
Sends a message to the net driver and returns status information from the driver. The two parameters are used for both.

Example :	SendNetwork(command, data);
	TOOLBOX(98, 100: commandH%, commandL%, dataH%, dataL%)
Parameters :	command (long) - address of the command; holds result after call
	data (long) - data for the command; data for the result
Errors :	$62FF - no net driver has been installed

SendNetwork commands:

Notes: (1) Commands marked with an asterisk (*) mark commands called automatically by DrawTools when required. (2) "Post" is used in the sense of PostEvent in the Event Manager: transmits a message on the network or to the driver.

0 - no command (use to poll the network)

1 - request the number of remote players

*2 - request the pseudo game driver address (returned in data) (used by SetNetDriver)

*3 - post a new local player (data=player#) (used by SetGameDriver)

4 - post a local player quitting (data=player#)

*5 - post local GetJoy result (data=device(byte1),axis(byte2),value(bytes3&4) (used by GetJoy)

*6 - post local GetFire result (data=result) (used by GetFire)

*7 - post local StillFiring result (data=result) (used by StillFiring)

8 - post abort game message (you can use it for whatever you want)

*9 - init the net driver (used by SetNetDriver)

*10 - shut down the net driver (used by DrawShutDown)

11-15 - reserved for future use

16-123 - application defined

124 - set address of where to receive incoming data (data=address) (for 125...127)

125 - prepare to transmit (data=player(low), number of blocks to be sent(high))

126 - block transmit(data=pointer to 256 bytes (a "block"))

127 - done transmit(data=player who should have received blocks)

128 - driver will begin displaying status information on the screen (use DrawTools' Print tools)

129 - driver will stop displaying status information

130-255 - net driver defined. With the Null Network Driver:

    130 - fake a new remote player (#2) beginning to play

    131 - fade a remote player (#2) quitting

>255 -reserved for future use

Results returned by SendNetwork:

Note: only 0 (null event) or errors should be returned during a block transmit or an information request, to avoid having to handle two things at once!

    0 - null event (nothing interesting happened)

    1 - abort game was received from a remote GS

    2 - a new remote player has started to play (data=player#)

    3 - an old player has quit playing (data=player#)

    4 - bad connection (can't find the network)

    5 - bad network error

    6 - network full (already 4 players playing)

    7-15 - reserved for future use

    16-124 - you received an application defined event of same number (data=other information)

    125 - prepare to receive transmission (data=player(low), number of 256 byte blocks (hi))

    126 - received 256 bytes of data (data=handle to data)

    127 - end of data (data=player who should have received data)

    >126 - reserved for future use

GetJoy ($4362)

Returns the position of the joystick along one axis. Horizontally, left (-2) through right (+2); vertically, top (-2) through bottom (+2). There must be a 3 microsecond delay between GetJoy calls.

Examples :	Position := GetJoy(Device, Axis);
	TOOLBOX(98, 67 : 0, Device%, Axis% ; Position%)
Parameters :	Position (word) - the joystick position, -2 ... 2
	Device (word)
	- 0 for internal joystick, or 1..4 for a game driver
	Axis (word) - 0 = horizontal axis ; 1 = vertical axis.
	- 2,3 - same, but for joystick #2 (*device 0 only*)
Errors :	Axis value ANDed with 3.
	$620F - device number out of range

GetFire ($4862)

Determines which joystick fire buttons have been pressed (but not held down) since last GetFire/StillFiring). The button addresses were taken from the November '90 issue of "8/16".

Examples :	Buttons := GetFire(Device);
	TOOLBOX(98, 72 : 0, Device%; Buttons%)
Parameters :	Buttons (word) - mask of fire buttons
	bit 0 = 1 => button #0 is depressed
	bit 1 = 1 => button #1 is depressed
	bit 2 = 1 => button #2 is depressed
	bit 3 = 1 => button #3 is depressed
	bits 4 - 15 are zero
	Device (word) - 0 for internal joystick, or 1..4 for a game driver
Errors :	$620F - device number out of range

StillFiring ($4D62)

Determines which fire buttons are being held down, whether or not they were during the last GetFire/StillFiring call. GetFire does not need to proceed a StillFiring call.

Examples :	Buttons := StillFiring(Device);
	TOOLBOX(98, 77 : 0; Buttons%)
Parameters :	Buttons (word) - mask of fire buttons
	bit 0 = 1 => button #0 is depressed
	bit 1 = 1 => button #1 is depressed
	bit 2 = 1 => button #2 is depressed
	bit 3 = 1 => button #3 is depressed
	bits 4 - 15 are zero
	Device (word) - 0 for internal joystick, or 1..4 for a game driver
Errors :	$620F - device number out of range

Miscellaneous Tools

GetQDT ($2062)

Returns the Quick Dispatch Table (QDT), a set of 16 JML instructions (64 bytes) to commonly used DrawTools routines. These are provided for assembly language programs that wish to avoid the overhead associated with tool calls. You must be in 16-bit native mode to execute the QDT routines. Jumping to a non-existed JML will cause unpredictable results, so check the toolset version before using GetQDT to ensure the JML's are available.

Preparing a quick dispatch table:

```
Draw        adrl 0 ; the quick dispatch table of 16, 4-byte JML entries
Draw48      adrl 0 ; in ORCA/M use i4
. . .
vector16    adrl 0
. . .
            PushPtr Draw
            _GetQDT
```

Using the quick dispatch table:

```
            LDA #ThePic       .
            JSL Draw
```

Register results after call:
   A - the result, if any
   X, Y, B, D - unchanged
   P - reflects the result, if any, else scrambled

The vectors are defined as:
Vector #1 - DrawTools 3.0 - Draw
Vector #2 - DrawTools 3.0 - Draw48
Vector #3 - DrawTools 3.0 - DrawOn
Vector #4 - DrawTools 3.0 - Draw48On
Vector #5 - DrawTools 3.0 - AnimatePixie (errors returned in A)
Vector #6 - DrawTools 3.0 - Rnd
Vector #7 - DrawTools 3.0 - Odds
Vector #8 - DrawTools 3.1 - WaitLine
Vector #9 - DrawTools 3.1 - ErasePixie (errors returned in A)
Vector #10 - DrawTools 3.1 - save interrupt space
Vector #11 - DrawTools 3.1 - restore interrupt space
Vector #12-#16 - reserved for future use

Vectors 10 and 11 backup DrawTools' direct page space. This allows you to call most DrawTools' functions from a RunQ task or another interrupt task. Alternately, you can use the scheduler. You will have to use these if an interrupt may occur during a DrawTools call: failure to do so may crash your program.

Examples :   For Merlin 16 : ~GetQDT #MyQuickDispatchTable
Parameters :   MyQDT (long) - location to save the copy of the quick dispatch table
Errors :   none

WorkCursor† ($4662)

Replaces the mouse cursor with the work cursor . Currently, the cursor is a pair of gears.

Examples :   WorkCursor (NumCalls)

                         TOOLBOX(98, 70 : NumCalls%)

Parameters :   NumCalls (word) - 0 = animate on every StillWorking, n = every nth

Errors :   none

† See WorkCursor2.

WorkCursor2 ($6862)

Same as WorkCusor, but works properly with accelerator cards.

Examples :   WorkCursor2 (NumTicks)

                         TOOLBOX(98, 70 : NumTicks%)

Parameters :   NumTicksIs (word) - 0 = animate every StillWorking, n = every n/60ths secs.

Errors :   none

StillWorking ($4762)

WorkCursor/WorkCursor2 must be called first. Checks to see if the work cursor needs animating. Use InitCursor if you want to restore the cursor to an arrow.

Examples :   StillWorking;

                         TOOLBOX(98, 71)

Parameters :   none

Errors :   none

Odds ($4462)

Returns TRUE the given percentage of the time. Percentages of zero or less are always FALSE; percentages of 100 or greater are always TRUE. This tool is accurate to about 2%.

Examples :   Boolean := Odds(Percent);

                         TOOLBOX(98, 68 : 0, Percent%; Boolean%)

Parameters :   Boolean (word) - the truth value

                         Percent (word) - the percentage of the time to be true.

Errors :   none

RND ($4562)

Returns a pseudorandom number between 1 and the specified limit. Limits of zero or less always result in zero.

Examples :   number := RND(limit);

                         TOOLBOX(98,69 : 0, limit% ; number%)

Parameters :   number (word) - the random number, 1...limit

                         limit (word) - the maximum random number (1...32767)

Errors :   none

NormalRND ($6562)

Returns a normally-distributed, or "bell-curved", pseudorandom number between 1 and the specified limit. The numbers are more likely to come from the center of the range than from the low or high ends of it.

Examples :	number := NormalRND(limit);
	TOOLBOX(98, 101: 0, limit; number)
Parameters :	number - the random number, 1...limit
	limit - the maximum random number (1...32767)
Errors :	none

HLoad ($2F62)

Handle LOAD. Loads a specified file into memory and returns a handle to it. (For those who like avoiding all those GS/OS details, like me.) The handle is left locked.

Examples :	DataHandle := HLoad(Path, FileType);
	TOOLBOX(98, 47 : 0, 0, PathH%, PathL%, FileType%; DataL%, DataH%)
Parameters :	Path (longword) - pointer to the GS/OS pathname
	FileType (word) - file type expected (or 0 for any type)
	DataHandle (long) - handle to the file data
Errors :	GS/OS and memory manager errors (file busy errors handled internally)
	$620B - FileType mismatch

HSave ($5462)

Handle SAVE. Saves the contents of a handle in a file. If a new file is created, the file type is the same as the FileType parameter, and the AuxType is 0. The handle is left locked.

Examples :	HSave(Path, FileType, DataHandle);
	TOOLBOX(98, 84 : PathH%, PathL%, FileType%, DataH%, DataL%)
Parameters :	Path (longword) - pointer to the pathname
	FileType (word) - file type expected (or 0 for any type)
	* type -1 is a special type: PNT/$0001, packed screen (used with SetBackground)
	DataHandle (long) - handle to the data to be saved
Errors :	GS/OS and memory manager errors (file busy errors handled internally)
	$620B - FileType mismatch

BarGraph ($4B62)

Draws a bar graph in a specified rectangle. The graph shows the percentage relationship between the "value" parameter and the max value in the graph record; values < 0% are treated as 0%; values >100% are treated as 100%. If the rectangle is larger vertically, the graph is drawn upward; if the rectangle is larger horizontally, it is drawn rightward.

Examples :	BarGraph(GraphRec, value);
	TOOLBOX(98, 75 : GraphRecH%, GraphRecL%, value%)
Parameters :	GraphRec (long) - pointer to a graph record

Graph record:

0-7	Graph	rectangle containing the graph
8,9	ForeCol	SolidPenPat value (-1 for current pen pat)
A,B	BackCol	SolidBackPat value (-1 for current back pat)
C,D	Max	maximum value for the graph
E-11	reserved	reserved; set to 0

Errors :	none

GetMHz ($6267)

Returns the current GS speed to the nearest MHz. (Also adjusts GetJoy so that it will operate properly at the current speed.)

Examples:	Speed := GetMHz;
	TOOLBOX(98, 103 : 0; Speed%)
Parameters:	· Speed (integer) - speed of the GS to the nearest MHz.
Errors:	none

PrintWindow ($6A62)

Sends a window, grafport or the screen to the printer. The Print Manager is automatically started, if necessary. No clipping is performed on overlapping windows. Also, you will need at least 32K free: PrintWindow saves the contents of the screen before showing the dialogs.

Examples:	PrintWindow(WindowPtr, Options);
	TOOLBOX(98, 106: WindowPtrH%, WindowPtrL%, Options%)
Parameters:	WindowPtr (long) - pointer to the window or grafport; if nil, prints whole screen
	Options (integer)
	bit 0 - if 1, shows the "Page Setup" dialog box
	bit 1 - if 1, shows the "Print" dialog box
	bit 2-15 - reserved; set to 0
Errors:	Print Manager errors
	Memory Manager errors

III. Appendices

Appendix A: DrawTools' Error Summary

Hex	Dec	Meaning
$0000	0	No error
$6201	25089	Too many libraries
$6202	25090	Sequence number out of range
$6203	25091	Invalid library ID
$6204	25092	The library is loaded
$6205	25093	Task signature missing/invalid
$6206	25094	Screen line out of range
$6207	25095	Task exists (or doesn't exist, depending on tool)
$6208	25096	SCB tasks are not enabled
$6209	25097	Library buffer tables full (currently, maximum 5 buffers, for 45K)
$620A	25098	Not enough memory in bank 0 for more buffers
$620B	25099	FileType Mismatch during a HLoad/HSave
$620C	25100	Sequence command mismatch (wrong command for this kind of pixie)
$620D	25101	Undefined sequence command in this version of DrawTools
$620E	25102	Pixie exists (or doesn't exist, depending on tool)
$620F	25103	Player/Device number out of range
$6210	25104	Game or Network Device not found
$6211	25105	Not a matted pixie
$62FF	25343	General error (consult tool description)
		- not implemented (ie. for Apple's two reserved tool numbers, #7 and #8)

Appendix B: Direct Page Usage

DP Addr	Label	Description
$0-3	SCRNPTR	Current drawing position, minus $2000
4-5	BASE_DP	Picture location in bank 0
6-9	PortPtr	Used by DrawMain & DrawShadow, the current grafport
A-D	GrafPtr	Ptr to QuickDraw II's pointer to the current grafport
E-F	MyID	Application's Memory ID, aux. type 15
10-23	Temp	Scratchpad space for DrawTools
24-27	LineTable	Ptr to QuickDraw II's line table
28-35	-	Used by fading and colour tools (don't modify)
36-41	-	Used by SCB Interrupt handler (don't modify)
42-43	StillFire	Bits set if fire buttons are held, %0..004321
44-45	FireMask	Bits true if fire button exists, $0..004321
46-55	PixAlloc	bit 7 - pixie allocated, bit 6 - pixie disabled
56-65	PixType	pixie types
66-75	PixVsMat	bit 7 - pixie visible, bit 6 - pixie matted
76-7B	-	Scratchpad space for Animation tools
7C-7D	FontCode	XOR of current font handle words
7E-81	GrafPort*	Ptr to pnloc field in current grafport
82-83	FontHeight	Current font height
84-85	LeftMargin	Left margin
86-87	RightMargin	Right margin
88-89	UtilTemp	Used for dereferencing
8A-8B	CurrentLib	Library ID for the current library
8C-FF	-	Misc. or future use (don't modify)

You may use any of the scratchpad space between DrawTools calls.

Appendix C: DrawTools and Other Toolsets

DrawTools should be compatible with all of the standard Apple toolsets. However, the following are a few things to notice.

1. DrawTools and ESP/FTA's SoundTools (TOOL219)
You cannot use the SCB Interrupt tools with the Soundtrack Tools.

2. Bit-mapped Graphics and QuickDraw II
a) **Coordinates** - DrawTools' coordinate system is identical to QuickDraw's 320 mode (0...319, 0...199). However, the coordinates are always global. Use the QuickDraw function LocalToGlobal when you are using windows/grafports to determine the proper coordinates.

b) **640 mode** - DrawTools drawing functions will work as you'd expect, creating 48x24 pictures instead of 24x24 pictures. The coordinates are always 0...319, 0...199, even if you are using QuickDraw in 640 mode. To determine the proper coordinates in a window/grafport, use the following (in Pascal): LocalToGlobal(WindowPoint);
WindowPoint.h := WindowPoint.h div 2;
Draw*whatever*At(WindowPoint.h, WindowPoint.v, picture_number);

c) **Mouse Cursor** - The drawing functions and screen scrolling functions operate directly on the screen, ignoring the mouse cursor. If you need a cursor on the screen, use HideCursor/ShowCursor.

d) **Clipping** - For speed, the drawing functions don't clip pictures to fit in the clipping regions of the current grafport (if you draw a picture, the entire picture is always drawn, even if it won't fit in a window).

3. Memory Manager
DrawTools uses auxID #15. When you shut down DrawTools, all memory allocated with aux ID #15 is disposed of (including any HLoaded handles).

Appendix D: Network and Game Drivers

What are Network and Game Drivers?

DrawTools lets you assign devices for up to four players. You can specify a device number when you call GetFire, StillFiring or GetJoy. Device 0 is always the GS joystick, but device 1 to 4 can be assigned to any device. By following the standards set in this addendum, your game (or other application) will be able to play with any device, allowing for even players on other GS's. All this is possible by what I call a game driver.

A driver in GS/OS is a piece of software that runs an input/output device, like a printer or a disk drive. A game driver is a piece of software that DrawTools uses to run an input device, typically for a game (hence the name). Game drivers are kept in a folder called **DT.Drivers**, located in the Tools folder on a boot disk. All your application has to do is use SFGetFile (the standard Open... dialog) to let the players select their drivers from that directory. You load them with the System Loader and tell DrawTools which game driver to use for which player, and the rest is done automatically.

If you want to go all the way and let players play on separate GS's, you'll need a net driver as well. This is a piece of software that DrawTools uses to communicate between separate computers, such as over a modem or an AppleTalk network. Using a net driver is a little more complicated than using game drivers alone, although DrawTools does a lot coordinating behind the scenes for you. You have to use a special tool called SendNetwork to send messages between the different GS's your program is running on. SendNetwork also returns to you status information about the other GS's, such as when a new player has started his computer and wants to join in, or when one of the existing players loses or wants to quit. Reading a player's device on another GS is done the same way as you would normally do, with GetJoy or the other joystick routines. If the player is not on your GS, DrawTools asks the Net Driver to find out the information for you.
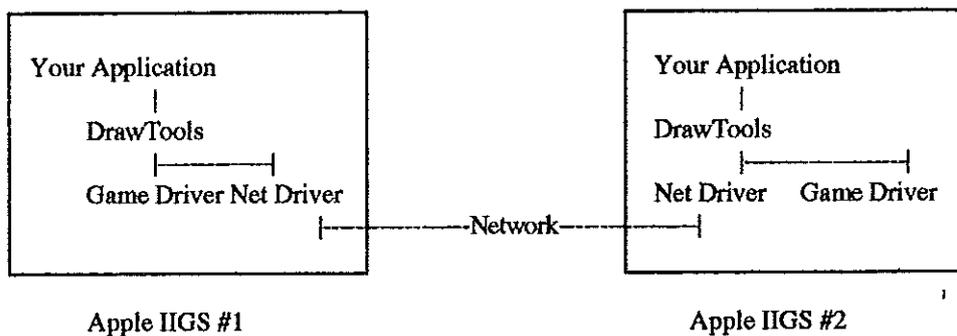
Figure 1 - How the Game and Net Drivers work together

I hope that by explaining the details here, that all the people that have more time than I do will get to work and start making game and net drivers. I set up the rules; somebody else makes the drivers. If you come up with a net or game driver, please send me a copy and a letter, and I'll try to market them with future disks. If you just want to use the net and game drivers, read on to find how how to set up your programs to support them. Located on the latest DrawTools' Disks is a folder called DT.Drivers, which you can copy into the Tools folder of your boot disk containing DrawTools. First, there is a sample game driver called **Joystick** that runs the GS Joystick. Second, there is a sample net driver called **Null.NetDriver** which mimics the some functions of a real net driver. (130 & 131 are special commands to mimic activities on a network for testing purposes - see SendNetwork in the reference.) Source files for the Merlin assembler are included in the folder. You can use these to test your program if you want to support game drivers, or net drivers and game drivers. As I mentioned previously, there is no reason "game" drivers have to be used in games. You might find it easier to write a game driver to operate a device like a flying mouse (the headset mouse used for the handicapped) than to write some kind of GS/OS driver (or whatever), and once written, such a driver can be used in any program supporting game drivers. The possibilities are enormous.

A Few Definitions:

A **device** is something used by a person to offer input to an application, such as a keyboard, joystick, Koala pad, or a microphone.
A **local** device is a device connected directly to a IIGS.
A remote device is a device connected indirectly to a IIGS, by an AppleTalk network, or a modem, or a SCSI port to another IIGS.

A **Game Driver** is a piece of software which operates or monitors a local device.
A **Net Driver** is a piece of software which is used by DrawTools to communicate over a network with remote devices.

What Does My Application Have To Do To Support Game Drivers?

What the Application does ...
1. You will have to load the game drivers desired by the players (using LoadDriver) into memory. The drivers should be located in the DT.Drivers folder in the Tools directory of the boot disk.
2. Use SetGameDriver (playerNum, DriverPtr) to install a driver for a particular person. One driver may be shared by more than one person (unless, of course, it's strictly a one person device, like a joystick — it's up to the players to chose devices that make *sense*).
3. When you use GetJoy, GetFire, or StillFiring, use the playerNum to specify a particular device.
4. Unload the driver when you are done.

What DrawTools does ...
DrawTools will invoke the appropriate game driver instead of reading the GS joystick. If no driver exists, garbage is returned by the call.

What the Game Driver does ...
The game driver reads the local device and returns the information requested to DrawTools, which hands it to your application.

What Does My Application Have To Do To Use Net Drivers?

What the Application does ...
1. Load the appropriate net driver into memory.
2. Use SetNetDriver(driverPtr) to install the net driver. The current version of DrawTools only supports one net driver; you can't play over two different networks at the same time.
3. When a player on a local device wants to start playing, use SendNetwork to inform the other GS (or GS's) that there is a new player. A message is returned if the GS's are full (DrawTools only supports 4 players at a time, even over a network).
4. Periodically invoke SendNetwork (eg. by placing it in your main loop) to let the net driver check on the network and keep up-to-date with the other GS (or GS's). This is called polling the network. If there are new players jumping into the game, or old players dropping out, SendNetwork will return the appropriate message. More details on the uses of SendNetwork are listed in the reference.

What DrawTools Does ...
If you use GetFire, GetJoy or StillFiring for a player on a remote device, DrawTools invokes the Net Driver and asks it to find the information, which it returns to your application.

What the Net Driver Does ...
The driver must handle the transmission and reception of data over the network. It takes care of identifying which player number on the local GS corresponds to which device on which remote GS. When a new player enters the

network, the net driver finds a free player number and reports it to your application for use with GetJoy, etc.

How to Create a Game Driver

1. File description: your driver must be stored in DT.Drivers folder in Tools folder of the boot disk. FileType: Generic Load File (type $BC) AuxType 1.

2. Header for you driver:

Offset	Name	Size	Description
0	EntryPt	(3 bytes)	BRL instruction to your driver
3	Name	(17 bytes)	Pascal string for the driver name
20	Creator	(17 bytes)	Your name or the name of your company
37	Version	(word)	Driver version (eg. $101 = 1.1)
39	DTVersion	(word)	Minimum version of DrawTools (eg. $301 = 3.1)
41	<reserved>	(8 bytes)	Zeroes
49	...		<your driver goes here>

DrawTools will call your driver with a JSL to the entry point. A = command, X = player #, Y = result of the command.

B & D registers must (naturally) be preserved. Place the result in A. Exit with a SEC and RTL.

The commands for game drivers are: 0 - init driver (called by SetGameDriver, return error code ($6210 or other) or else 0) 1- GetJoy (Y=axis, called by GetJoy) 2 - GetFire (called by GetFire) 3 - StillFiring (called by StillFiring)

How to Create a Net Driver

1. File description: must be stored in DT.Drivers folder in Tools folder of the boot disk. FileType: Generic Load File (type $BC) AuxType 2.

2. Header for you driver:

Offset	Name	Size	Description
0	EntryPt	(3 bytes)	BRL instruction to your driver
3	Name	(17 bytes)	Pascal string for the driver name
20	Creator	(17 bytes)	Your name or the name of your company
37	Version	(word)	Driver version (eg. $100 = 1.00)
39	DTVersion	(word)	Minimum version of DrawTools (eg $301 = 3.1)
41	<reserved>	(8 bytes)	Set to 0
49	...		<your driver goes here>

DrawTools will call your driver with a JSL to the entry point. A is the command, X = data (low), Y = data (high).

B & D registers must be preserved. Return with the result in A, and any data in X,Y. Exit with an RTL.

For the commands, see the reference under SendNetwork. You will need a pseudo game driver for DrawTools to call when it wants information for a local device. For your game driver, design it to be called like a regular game driver, *except* return with a CLC (not SEC) and RTL. This will make sure DrawTools won't send the results of the GetJoy/etc. back to you (posting local events).

Appendix E : Using PicEd 3.0

Besides the library converter utility, there is a utility called PicEd that helps you to create libraries of the bit-mapped pictures that DrawTools' works with. PicEd was written in TML Pascal II, v1.1.

When PicEd is started, there is a large grid of 24x24 black blocks to the right of the screen. This is a zoom (fat pixels) view of the current picture. Using the mouse, you can change the blocks to different colours. While you are editing a picture, the changes you make are TEMPORARY until you select the EDIT button. This way, if you make a mistake, you can always revert to the original copy of the picture and start again.

When blocks in the zoom view are changed, these changes are reflected on a series of pictures in the top-left corner of the screen. The large picture is a view of how the picture would look if it were drawn with the Draw48 call. To the left of this picture are three smaller ones. The one on the far left is drawn with Draw. The one in the middle is drawn with DrawOn (that is, matted) on a red background. The one on the right is used when animating.

There is a palette of colours to the left of the zoom view. You can change the colour you are sketching with by clicking on a new colour. The new colour is outlined in black.

Below the palette is a series of buttons:

QUIT - this stops PicEd. It gives no warnings, so make sure your work is saved.

CLR - clears the zoom view to black.

EDIT - saves the current picture in the library, and selects another for editing

When EDIT is first clicked, PicEd gives you three options: (S)ame - save the current picture to library position it was edited from; (D)ont - don't save the current picture in the library; (N)ew - save the current picture to a new position. If you pick new, you will be asked for a new position (0...31).

LOAD - loads a library of pictures from disk.

SAVE - saves a library of pictures to disk. Pressing Return will use the LOAD name as the default.

MASK - calls GenMask to create a simple matting mask. Normally, this mask is stored immediately after the picture it was created for.

The PAL and ANI buttons are special buttons which cause a new set of buttons to appear on the screen. The PAL (palette) buttons are:

PAL - let's you select one of 16 palettes to use, palette 0 being the default palette of colours used by QuickDraw. If you change any of the palettes (besides palette 0), the information is saved in the file PicEd.dat, and the palettes will be reloaded the next time you run PicEd. Some of the palettes are predefined as the 640 colours, the standard IBM VGA colours, metallic and rainbow colours.

COL - change a colour in the current palette.

FADE - brightens or dims a colour by using the FadeColour tool.

BLND - blends two colours together to produce a third by calling the BlendColour tool.

The ANI (Animation) buttons as as follows:

DONE - you are finished animation. Gets you out of animation mode and restores the other buttons.

SEQ - define an animation sequence. If you want to animate a set of pictures in the current library, select this button, then type in each picture you want to animate, in order. Then type 255 and type in the position in the sequence you want to loop back to (ie. 0 = first position, 1 = second, etc). To animate the first 3 pictures over and over, you'd type: 0 then 1 then 2 then 255 then 0.

GO! - animates the sequence you typed in. Hold down the mouse to stop. From left to right, the pictures are drawn: 1) as a matted pixie on a red background, 2) as a pixie that is not matted, 3) as a 48x48 picture (by Draw48).

Appendix F : Using Library Converter 1.2

Lib.Converter, the library converter, is a utility that lets you translate a picture library template into a DrawTools picture library. A template is simply a super hi-resolution screen with the 32 pictures of a picture library laid out for you to edit with any paint program. Keep in mind that the template must be saved as a super hi-resolution screen and not as one of the other picture formats, such as Apple Preferred.

Convert Template to Library ... (Command-Oo): Select this to convert a template to a picture library. Lib.Converter will ask you which template you would like to convert. During the conversion, the template pictures are displayed on the desktop. Once the template is converted, Lib.Converter will ask you what name you would like to save the picture library as.

Display a Template ... (Command-Dd): Select this to display the pictures in a template on the screen. The colours may differ from the original template.

Print a Template ... (Command-Pp): Select this to print a template to the printer. Lib.Converter uses PrintWindow to print the entire screen (including the pictures).

Convert SHR Screen to SetBack ... (Command-Bb): Select this to convert a super hi-resolution screen to a packed super hi-resolution screen, the format used by SetBackground and SetBackground2.

Pack (Command-Pp): When Pack is checkmarked, the template you convert with "Convert Template to Library" will be packed.

Appendix G : Changes Since DrawTools 3.0

1. New ORCA/M macros.
2. CLS now works with a visible cursor.
3. NormalRND no longer returns a uniform distribution.
4. NEW WorkCursor2: WorkCursor that works with accelerator cards.
5. New QDT Vectors:

      #8 -> WaitLine   #9 -> ErasePixie

      #10 -> Save interrupt space   #11 -> Restore interrupt space

6. HLoad waits until a file is not busy.
7. HLoad now works with files larger than 64K.
8. Change status command for fine pixies now works.
9. The library limit has been increased to 24 from 16.
10. Library Converter has been updated to version 1.1. Requires System 6.0.
11. SetGameDriver no longer crashes and it returns error $62FF properly.
12. NEW ErasePixie: A more convenient form of WipeOn.
13. NEW EraseAllPixies.
14. NEW LoadDriver: Loads a game or net driver.
15. NEW UnloadDriver.
16. BarGraph supports 16 colours for the forecolour, if you are using System 6.0.
17. SetNetDriver returns error $62FF properly.
18. NEW SetBackground2: SetBackground with more options.
19. NEW Keypad.Drvr & Keybrd.Drvr: game drivers for the Apple IIGS keyboard.
20. NEW ResetBuffers: Clears the bank 0 drawing buffers.
21. NEW PrintWindow: Print the contents of a window or the screen.
22. New self-running Micol Advanced BASIC demo.
23. WaitLine is now more accurate: interrupts are suspended to ensure prompt response. (This was the problem that made Quest for the Hoard™ sluggish when many inits were installed.)

Tool Index (in alphabetical order)