NOTE: This Technical Note has been retired. Please see the Technical Notes page for current documentation.

Technical Note TN1053

QuickDraw GX GraphicsBug: Description, Uses &Limitations

CONTENTS

Important for all Apple Printing and Graphics Developers:

The information in this Technote is still relevant up to and including Mac OS 7.6 with QuickDraw GX 1.1.5. Beginning with the release of Mac OS 8.0, however, Apple plans to deliver a system which incorporates QuickDraw GX graphics and typography **only**. QuickDraw GX printer drivers and GX printing extensions will **not** be supported in Mac OS 8.0 or in future Mac OS releases. Apple's goal is to simplify the user experience of printing by unifying the Macintosh graphic and printing architectures and standardizing on the classic Printing Manager.

For details on Apple's official announcement, refer to </technotes/gxchange.html>

This Technote discusses GraphicsBug, the GX debugger application. It provides a description of the history of GraphicsBug, as well as an explanation of how GraphicsBug can best be used by GX developers. This Note also includes a list of all the currently known bugs in GraphicsBug.

This Note is intended for Macintosh QuickDraw GX developers who are developing applications with QuickDraw GX version 1.1.3 or earlier.

Updated: [July 1 1996]

About GraphicsBug

GraphicsBug is a quirky utility that provides a Macsbug-like view of Quickdraw GX applications. Although it may look like an application itself, don't let the menus and windows fool you; GraphicsBug is best used sparingly as a command-line debugger, not a cut-and-paste text program. GraphicsBug is aptly named: it's full of bugs. Still, it can be an indispensable tool when GX isn't working the way you expect.

GraphicsBug in Action

One simple task GraphicsBug performs well is confirming that your GX Graphics code does what you think it does. After you've executed the code, you can determine:

- That the shapes, styles, inks, transforms and so on were created as you expect.
- That no errors, warnings or notices were posted.
- How much memory the graphics objects require.
- If your code has any GX-related memory leaks.

Starting up GraphicsBug

GraphicsBug for QuickDraw GX 1.1.3 is available on the Apple web site.

> **Important:**
> You'll want to make sure that GraphicsBug is the same version as the GX INIT you're using. GraphicsBug will work with either the regular install of GX and/or the Graphics debugging INIT, but it may blow up if neither is installed, or if the versions don't match.

When GraphicsBug launches, it opens an untitled window, as shown in Figure 1.

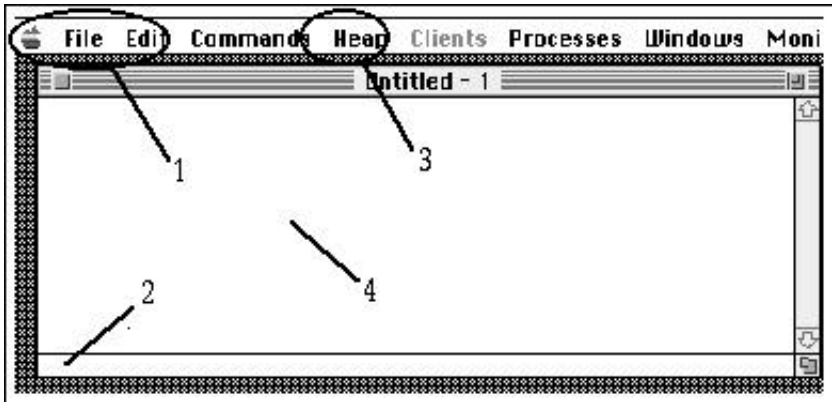

Figure 1. A GraphicsBug window

GraphicsBug appears to be an ordinary application with traditional menus (1 in Figure 1). (Avoid these.) The real action happens in the Command line (2 in Figure 1) -- most commands are entered here. Always start by switching to your GX application's heap. You can choose it in the Heap menu (3 in Figure 1), or by using the Heap eXchange (hx) command.

GraphicsBug commands can be extremely terse. For instance, the following selects only the GX application that begins with the letter 'm':

```
hx m
(turned "m" into "My GX App")
Heap set to 009c3f9c  "My GX App".
```

The content pane of the window (4 in Figure 1) is used for GraphicsBug's response to commands.

Note:
If you're not using the debugging INIT (if there isn't an INIT in Extensions called GXGraphics), you won't be able to select the heap by name.

You can find the address of your application with the LC command, which stands for List Clients:

```
lc
  Client        Process         other      &ap      ApHeap      Name
 00a59e54  0000000000002006  00a59edc  009c3e34  009c3f9c  "My GX App"
```

Set the heap to the address under the ApHeap column:

```
hx 009c3f9c
Heap set to 009c3f9c  "My GX App".
```

GraphicsBug Basics

There are only a few commands you'll need to debug most graphics and printing applications. The remaining commands are mostly useful for GX engineers and those curious about the inner workings of GX. Use any command with care; it's best to have work in progress in other applications saved, and to have the real Macsbug installed. While GraphicsBug does not have any demonstrated circumstances under which it corrupts memory, various commands can cause a crash.

Listing GX Objects

Once you've selected your application heap, you're ready to explore the GX heap. To get a list of the objects in a GX heap, use hd, for "HeapDump". For instance, dumping a GX application that has created one `gxLineType` shape yields:

```
hd
 Start    Length  delta   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00abdd70 0000001c+00   d       00abdd8c       b        heap part block
00abdd8c 00000118+00   d       00000000       b        heap header block
00abdea4 0000024e+02   d       00000000                freeFileList
00abe0f4 00000014+00   d       00236838                fontList
00abe108 0000004c+00   i       00b53c10                line
00abe154 00000064+00   i       00b53c0c                transform
00abe1b8 000000c4+00   d       00000000                port
00abe27c 0000003c+00   i       00b53c08                full
00abe2b8 000000b4+00   i       00b53c04                style
00abe36c 00000038+00   i       00b53c00                ink
00abe3a4 0000004c+00   i       00b53bfc                line
00abe3f0 00095720      f       00000000                free block
00b53b10 00000110+00   d       00abdd8c       b        master pointer block
00b53c20 00000010+00   d       00abdd8c       b        heap trailer block
          Total Blocks                 Total of Block Sizes
Free      00000001    #        1       00095720    #   612128
Direct    00000003    #        3       00000328    #      808
Indirect  00000006    #        6       00000224    #      548
Sub Heaps 00000000    #        0       00000000    #        0
Heap Size 0000000a    #       10       00095ec0    #   614080
```

Heap dumps include all blocks allocated by GX, including undocumented internal blocks.

The details of what's in a heap dump are covered in the next section, but there are a few generalities worth noting:

- GraphicsBug likes to show numbers in hexadecimal; this is also the default base for entered numbers.
- Hex numbers in the content area are zero padded; decimal numbers are space padded and preceded by a number symbol.
- The previous example heap looks like a Macintosh heap, but it's not. This may change in future versions of GX, but for versions up to 1.1.3, the heaps displayed by Macsbug look similar to but are different from Macintosh heaps. This means that if you have a Macintosh handle, you'll want to examine it with Macsbug or a tool such as Metrowerk's Zone Ranger. If you

have a GX object reference, you'll want to examine it with GraphicsBug.

Look at Them Shapes

The easiest way to start a GX heap survey is to get a list of all of the shapes allocated by the application. A number of the commands can have a "shape" qualifier: this restricts the blocks listed to the shapes the application has created. For instance, this code:

```
gxLine data = {{0, 0}, {ff(125), 0}};
gxShape line1 = GXNewLine(&data);
```

generates:

```
hd shape
 Start    Length  delta   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00a45ac8 0000004c+00   i       00adb5d0                line
          Total Blocks            Total of Block Sizes
Blocks    00000001  #        1    0000004c    #       76
```

Generating one shape in the source code resulted in one shape in the dump; perfectly reasonable. But look what this command tells us:

```
hd line
 Start    Length  delta   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00a45ac8 0000004c+00   i       00adb5d0                line
00a45d64 0000004c+00   i       00adb5bc                line
          Total Blocks            Total of Block Sizes
Blocks    00000002  #        2    00000098    #      152
```

The other line is the default shape that's used internally by GX. There are a host of circumstances where GX creates internal shapes; the "shape" attribute winnows those internal shapes out.

There are a host of ways to view the contents of a GX object. In this case, the da command (for "display all") is handy:

```
da line shape
Displaying line gxShape from 00a45ac8
  devShape        nil
  owners            1
  seed              0
  flags             0
  attributes no attributes
  gxStyle      00a45c78
  gxInk        00a45d2c
  gxTransform  00a45b14
  tagList          nil
  cacheList        nil
  geo.flags          0
  fillType   openFrameFill
{     0.0000,     0.0000}  {   125.0000,     0.0000}
```

Like hd, the da command can be used with or without modifiers. More than one modifier acts as an additional qualifier; only objects that meet all the conditions are listed.

You can use the Display Memory (dm) command to display an object if you know the address. (This was the first command created for GraphicsBug.)

```
dm 00a45d64 t
Displaying line gxShape from 00a45d64
  devShape        nil
  owners            1
  seed              0
  flags      isDefaultShape
  attributes no attributes
  gxStyle      00a45c78
  gxInk        00a45d2c
  gxTransform  00a45b14
  tagList          nil
  cacheList        nil
  geo.flags          0
  fillType   openFrameFill
{     0.0000,     0.0000}  {     0.0000,     0.0000}
```

The 't' stands for "typed". Another way to display an object is to option-double click on the address.

The address you have may either be in the object or just associated with it. You can use the WHere command to determine if the address is in any object.

```
wh 00a45aa8
Address 00a45aa8 is in the heap at 00a4567c "My Graphics App".
It is 00000000 bytes into this heap block:
 Start    Length  delta   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00a45aa8 000000c4+00   d       00000000               port
```

The Find command can be used to determine what objects are associated with an address. Here's a heap dump fragment that shows the address of an ink.

```
 Start    Length  delta   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00a45c5c 00000038+00   i       00adb4f0               ink
```

You can use Find to return objects that have references to the ink. Type 'f', then hold down the Command key and click on the "Mstr Ptr" value to copy it to the command line. (If the Command shortcut is stubborn, try clicking in the command line before clicking on the number to copy.)

```
f 00adb4f0
 Start    Length  Δ   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00a459f8 0000004c+00   i       00adb500                line
 00a45a10  00ad b4f0 00ad b4fc 0000 0000 0000 0000  ⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗
00a45c5c 00000038+00   i       00adb4f0                ink
 00a45c58  00ad b4f0 0000 0000 0000 0100 0000 0003  ⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗
00a45c94 0000004c+00   i       00adb4ec                line
 00a45cac  00ad b4f0 00ad b4fc 0000 0000 0000 0000  ⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗⊗
          Total Blocks           Total of Block Sizes
Blocks    00000003  #        3    000000d0    #      208
```

In this example, there are three references to the ink: the line shape created by the application, the internal default line, and the ink itself. There may be more shapes that use this ink that are not shown by this command. GraphicsBug only searches the RAM-resident graphics heap and does not search the disk-based backing store (where object unloaded to disk reside).

You can qualify Find just like Heap Dump. Here, we request all user allocated shapes that refer to the ink in question:

```
f 00adb4f0 shape
 Start    Length  Δ   Typ Busy Mstr Ptr Temp TBsy Disk   Object
00a459f8 0000004c+00    i       00adb500                line
 00a45a10  00ad b4f0 00ad b4fc 0000 0000 0000 0000  ????????????????
           Total Blocks            Total of Block Sizes
Blocks    00000001  #        1    0000004c    #       76
```

Note:
The names that GraphicsBug uses for blocks it slightly different from the public object types. For most names, you can drop the initial "gx" and the trailing "Type" if any, and get the internal block type. For instance, a `gxPointType` internally is a point, and a `gxTransform` is a transform. In the case of `gxViewPort` and `gxViewDevice`, the internal blocks are named port and device. GraphicsBug will attempt to complete partial names, so you don't have to remember whether to type polygons or polygon; poly will do.

This completes the indispensable portion of GraphicsBug. If you forget what you've learned so far, just remember to type '?'. This returns a summary of the GraphicsBug commands.

GraphicsBug Reference

This section spells out what the commands do and how GraphicsBug works.

GraphicsBug and QuickDraw GX

When GraphicsBug launches, it also establishes communication with the rest of GX via a special debugging interface. This means if GX is not around, GraphicsBug may crash.

It's not necessary to have a GX application running to launch GraphicsBug. With only the Graphics debugging INIT installed in your system, the Finder will not be a GX client and GX Printing will not be available, but calls to GX graphics, fonts and text are still available.

As a default, GraphicsBug selects the GX system heap. Analogous to the GX application's heap, the GX system heap has nothing to do with the Macintosh system heap; it is only the container for storing GX objects common to all applications, like the gxViewDevice associated with the screen.

The Zen of Being Up-To-Date

In different releases of GX, the memory blocks visible to GraphicsBug can and do change: note that some of the blocks listed in the previous examples don't have real object names. That's because new internal types were added to GX after GraphicsBug was last revised. This may sound like GraphicsBug is "out of date." In fact, there are two ways GraphicsBug can be out of date. When GraphicsBug is compiled, it includes some part of the GX source base; if it was compiled against a different source base than the one you're running, GraphicsBug is likely to crash and burn on launch or soon afterwards. Also, GraphicsBug has tables of types that are maintained by hand; if these tables are out of date, then some block types can't be displayed. This is less serious, since the block types of interest to most developers have templates which are complete and accurate.

Common Command Summary

GraphicsBug allows you to examine the graphics heap in detail. Here are the commands you're most likely to use:

Heap eXchange

```
HX addr | <heapname>
```

Switches to the heap containing addr, or named <heapname>. This command works reasonably well from the command line or from the menu. It's OK to quote the <heapname>. If there's no debugging INIT, you can use the LC command to find the heap address of your GX application. You'll want to use HX to select the correct heap before using the other GraphicsBug commands.

Display All

```
DA [<type> ... ] [shape]
```

Displays all blocks in the heap, or all that match parameters.

Display Memory

```
DM addr t
```

Display memory from addr using the appropriate template for that type. Or, option double click on the address to display memory using a template.

Find

```
F addr [<type> ... ] [shape]
```

Finds references to addr in the heap parts that match parameters.

Heap Dump

```
HD [<type> ... ] [shape]
```

Dumps the heap, or the heap parts that match parameters.

WHere

```
WH addr
```

Displays the block containing addr.

Miscellany

Simple commands like cut-and-paste do not entirely work. GraphicsBug does a terrible job of maintaining the current selection, for instance. There are a few reliable techniques worth knowing:

- A command-click on a number in the content pane will copy it to the command line. Click in the command line before the command-click.
- You can get a summary of the commands available with "?".
- Reset the content area of GraphicsBug by clicking in the content area, pressing A (Select All) and hitting the delete key.
- The content area will initially record only 32000 bytes. To save a lot of information, increase "Window Buffer Size" in the Preferences Dialog, close the current window, and open another one. If opening the window fails, quit, increase the size of the GraphicsBug heap, and relaunch.
- Option-double-click on an address to display memory as a type.
- Command click on a number to copy it to the command line.
- Use 'shape' as an argument to DisplayAll, Find and HeapDump to display all client-owned shapes.

Uncommon Commands and Command Options

You can stop here. Really. But if you are an information junkie, here are some additional options and commands that may come in handy from time to time.

Somewhat Useful Commands

DisplayMemory Macsbug-style

```
DM [addr [length]]
```

The `DisplayMemory` command can produce ordinary memory dumps.

```
dm 900 20
Displaying memory from 00000900
 00000900  183e ff20 010c 1f70 010b e93a ffff ffff  ◊>◊ ◊◊p◊◊:◊◊◊◊
 00000910  0b47 7261 7068 6963 7342 7567 0010 880e  ◊GraphicsBug◊◊◊◊
```

DisplayMemory attempts to be the same as Macsbug. Unlike Macsbug, however, pressing return doesn't display more of the same address.

ERror

```
ER number
```

Displays the error name that matches this number. It's easiest if the number is hexadecimal, but if you precede the number with a number sign, you can enter decimal as well.

```
er ffff96EC
graphicsWarning: contour out of range
er -#26900
graphicsWarning: contour out of range
```

Putting the minus sign in front of the number won't work:

```
er #-26900 : (may be a Macintosh file system error)
```

What #-26900 evaluates to is anybody's guess.

Find Within Memory Range

```
F addr [number [start [end]]]
```

You can refine the Find command by specifying the number of items to find, and the address range of those items. The number of items doesn't have any effect, but the start and end range work OK.

HeapCheck

```
HC
```

If you have the rare bug that corrupts the GX heap, you can use HeapCheck to isolate the offending code.

Special Block Qualifiers

```
DA [bu(sy) di(rect) fr(ee) i(ndirect) t(emp) u(n)b(usy)u(n)l(oaded)]
F[bu(sy) di(rect) fr(ee) i(ndirect) t(emp) u(n)b(usy)u(n)l(oaded)]
HD[bu(sy) di(rect) fr(ee) i(ndirect) t(emp) u(n)b(usy)u(n)l(oaded)]
```

You can qualify HeapDump, Find, and DisplayAll with some implementation-dependent parameters. As of this writing, all blocks are either direct, indirect or free. Indirect blocks are shapes, styles, inks, transforms, color sets and color profiles. All other blocks are direct blocks. An indirect block always has a master pointer; a direct block has a single owner containing the pointer to that block. Unlike the Memory Manager's pointer blocks, direct blocks can be relocated.

Internally, blocks may be locked down; the bu parameter lists these busy blocks. You can explicitly lock busy blocks by calling GXLockShape. GX may create temporary blocks during an operation; the t parameter lists temp blocks. Normally, you'll never see

temp blocks, but while debugging a callback function, it's possible that you'll see a temp block in the GX heap. If you see a temp block outside of a GX call, you're likely looking at a GX bug.

ValidateAll

```
V  [addr]
```

Validate all blocks (no parameters) or validate a specific block.

ValidateAll does a better job than HeapCheck in looking for block corruption; while HeapCheck can only check the length of blocks and some simple pointers and flags, ValidateAll can check the flags and pointers internal to the blocks that GX allocates. ValidateAll with no parameters checks all blocks for valid contents. Unfortunately, ValidateAll with a parameter doesn't work correctly.

Useless Commands

These commands you'll likely never need, but for the sake of completeness, here they are. The explanations that follow are sparse, but after all, the commands are practically useless.

DisplayVersion

```
DV
1.1.2  (built on Apr 14 1995 at 19: 15: 40)
Graphics gestalt version0x00010100
```

The only thing that DisplayVersion has going for it is that you'll get a better idea of when GraphicsBug was last revised than from looking at the creation date.

Flatten

```
FL addr [filename]             Ex.: FL 0x3321A "flat shapes"
```

Display the stream produced by flattening this shape

Flatten performs the same work as GXFlattenShape. Here's what the output of Flatten looks like, given a reference to a line:

```
fl 009c4388
newObject; size: #2 (03)
headerType; byte compression (80)
version == 00010000; flags == fontListFlatten | fontGlyphsFlatten
(01 03)
newObject; size: #6 (07) [1]
fontNameType; no compression (2f)
(04 c8 8e 84 00 00)
newObject; size: #0 (01) [1]
styleType; no compression (28)
newObject; size: #0 (01) [1]
inkType; no compression (29)
newObject; size: #0 (01) [1]
transformType; no compression (2a)
newObject; size: #4 (05)
lineType; byte compression (83)
(00 00 7d 00)
newObject; size: #0 (01)
trailerType; no compression (3f)
```

The numbers in parentheses are data. The numbers in brackets are reference counts. The numbers after the number sign are stream data sizes, not counting the stream data two byte header. The "no/byte/word compression" refers to whether the actual data is larger than the shown data. For instance, the byte data after `lineType` is converted into four longs by sign extending the byte to a 16 bit word, then padding the word with 16 bits of zeros to represent a Fixed.

Only data that differs from the INIT default values is written; that's why the style, ink and transform in this example have no data. The line and its companions can be represented in just 21 bytes.

You'll see more of this in *Inside Macintosh: GX Environment and Utilities* . If you specify a filename, GraphicsBug will save the flattened object in binary form in the file. The file type will be "flat". You can pass this file to the `UnFlatten` command, described later in this Technote.

Graphics Globals

```
GG
```

Display graphics globals

This command usually returns the wrong globals. To get the correct graphics globals, follow these steps instead:

1. Use ListClients to get the gxClient address.

```
lc
  Client        Process         other      &ap      ApHeap      Name
 00ae0974  0000000000002006  00ae09fc  00a4a954  00a4aabc  "My GX App"
```

1. Use DisplayMemory (or option double click on the address) to display the client.

```
dm 00ae0974 t
clientRecord at 00ae0974:
  nextClient            nil
  heapStart             nil
  heapLength       00000000
  attributes       00000000
  otherGlobals     00ae09fc
  graphicsGlobals  00a4a954
  graphicsHeap     00a4aabc
  owner            0000000000002006
  users       00000000
```

1. Use `DisplayMemory` (or option double click on the address) to display the graphics globals.

```
dm 00a4a954 t
graphics globals at 00a4a954:
  backingStore        00a4abd4
      highest write   00000512
  matchingData             nil
  hitTestSlabGlobals       nil
  portList            00a4aee8
  deviceList               nil
  nextPortOrder       00000002
  nextDeviceOrder     00000001
  nextViewGroup       00000003
  windowList               nil
  flatInfo                 nil
  flatSpool                nil
  drawShapes:
  defaultShapes:
          line  00ae092c
  defaultStyle        00ae0934
  defaultInk          00ae0930
  defaultTransform    00ae093c
  defaultBitmapSets:
  defaultPort         00000001
  defaultProfile      00000000
  fontList            00a4ae24
  defaultFont         00000000
  translatorPtr       00000000
  bmDiskCache         00000000
  alreadyHaveFontList    false
  alreadyHaveFontFamilies false
  groupList                nil
```

HeapTotal

```
HT
```

`HeapTotal` returns the number and amount of direct, indirect and free blocks.

```
ht
Totaling the heap at 00a4aabc (My GX App heap).
          Total Blocks                Total of Block Sizes
Free      00000001    #        1      00095720    #   612128
Direct    00000003    #        3      00000328    #      808
Indirect  00000006    #        6      00000224    #      548
Sub Heaps 00000000    #        0      00000000    #        0
Heap Size 0000000a    #       10      00095ec0    #   614080
```

`HeapTotal` works, and is accurate. Unfortunately, there are few practical examples where the results are important. Because of the way GX can use MultiFinder temporary memory and the disk to store information, the result of the `HeapTotal` can be deceiving.

HeapZones

```
HZ
```

Lists the known heaps.

If you forget the name of your application (and you're running the debug init), this will help refresh your memory.

Without the debug init, only addresses will appear in response to this command.

```
hz
002b92b0 start (system.graphics heap)
002eb284   end
00a4aabc start (My GX App heap)
00ae0954   end
```

`ListClients` is a slightly more useful alternative command.

InitGlobals

```
IG
```

Displays INIT globals.

```
global handle: 0x000cc764  global pointer: 0x000e30f0
  initFileName               "GXGraphics"
  initVRef                   0xffff
  initDirID                  0x00001592
  rsrcFileRef                0x0000
  debuggerInfo               0x0014d746
  memoryDispatcher           0x00000000
  dispatchSetTrapAddress     0x0006ca98
  dispatchGetTrapAddress     0x0003598e
  dispatchDispatchText       0x0015be3c
  dispatchDispatchLine       0x0015be44
  dispatchDispatchRect       0x0015be4c
  dispatchDispatchRRect      0x0015be54
  dispatchDispatchOval       0x0015be5c
  dispatchDispatchArc        0x0015be64
  dispatchDispatchPoly       0x0015be6c
  dispatchDispatchRgn        0x0015be74
  dispatchDispatchBits       0x0015be7c
  dispatchDispatchComment    0x0015be84
  patchPictTrap              0x0015be94
  originalMaxApplZone        0x4080d2dc
  originalInitGDevice        0x000d28b4
  originalSetDeviceAttribute 0x40828000
  originalSetEntries         0x000182d2
  originalBringToFront       0x000acfc4
  originalCalcVBehind        0x000ac158
  originalCleanupApplication 0x000d611a
  activeClientAddress        0x00149790
  activeProcessAddress       0x00149798
  originalLaunch             0x00026cee
  originalOSDispatch         0x0025b820
  originalTempNewHandle      0x0025b820
  activeClientAddress        0x00149790
  activeProcessAddress       0x00149798
  sysHeapAddress             0x0014e848
  graphicsA5                 0x0015747a
  rootCallMade               0x0000
  insidePrinting                      0
  systemPatchesInstalled              1
```

These globals are used by all GX clients. The main use of InitGlobals is to reveal which traps GX patches.

ListClients

```
LC [process]
```

Lists the known graphics clients.

```
lc
  Client        Process         other      &ap      ApHeap      Name
 00ae0974  0000000000002005  00ae09fc  00a4a954  00a4aabc  "My GX App"
```

ListClients shows how GX connects a gxGraphicsClient to the graphics heap, the Process Manager and the internal client record.

ListProcesses

```
LP
```

Lists the known processes, with or without a graphics client.

```
lp
 Process     Process #       Active      Name
                             Client
00092594  0000000000002005  00ae0974  "My GX App"
00289d24  0000000000002004  00000000  "MW Debug/MacOS 1.4"
0000c854  0000000000002003  00000000  "GraphicsBug"
0031c0b0  0000000000002002  00000000  "AppleWorks"
00290444  0000000000002001  00000000  "CodeWarrior IDE 1.4"
00019674  0000000000002000  00000000  "Finder"
001544f0  0000000000000000  00000000  "null process"
```

ListProcesses unveils that this Technote was written in AppleWorks while using an example program called My GX App under Metrowerks to generate some GX objects, which were viewed with GraphicsBug.

OtherGlobals

```
OG
```

OtherGlobals attempts to display other (generic, non-graphic) globals used by GX.

Unfortunately, this is another command that doesn't work directly. You can get the correct result though ListClient instead. In this example, a shape is accidentally disposed twice. That causes the other globals to look like:

```
lc
  Client        Process        other      &ap     ApHeap      Name
 00d41214  0000000000002009  00d4129c  00a672e4  00a6744c  "My GX App"
dm 00d4129c t
generic globals at 00d4129c:
  lastWarning             : 00000000
  lastError               : shape access not allowed
  lastNotice              : 00000000
  stickyWarning           : 00000000
  stickyError             : shape access not allowed
  stickyNotice            : 00000000
  userError               00000000()
  userErrorRef            00000000
  userWarning             00000000()
  userWarningRef          00000000
  userNotice              00000000()
  userNoticeRef           00000000
  ignoredWarnings               0
  ignoredNotices                0
  randomSeed              00000000 00000000
  validation                    0
  checkLeafs                    0
  checkRoots                    0
  validationProcedure     00000000
  validationArgumentNumber      0
  validationArgumentValue 00000000
  currentProcAddr         00000000
  currentProcName         (none)
  typeName                (none)
  validationInProgress    false
  foundError              false
  userDebug               00000000
  debugReference          00000000
```

A more useful way to find errors is to use the `GraphicsDebugLibrary` and call `SetGraphicsLibraryErrors()` at the beginning of your application. If you're working with an application you didn't write, however, this will do.

Quit

```
Q
```

Quits out of GraphicsBug.

Unflatten

```
UF filename [page number]
```

`UnFlatten` is the companion to `FLatten`. It can display the contents of a file saved by `FLatten`, or a printer spool file. Since the dumps of printer spool files can be huge, you can also specify a page number to `Unflatten`.

```
UF "save me"
(80) headerType; byte compression
(01 03) version == 00010000; flags == fontListFlatten | fontGlyphsFlatten
(07) newObject; size: #6
(2f) fontNameType; no compression [1]
(04 c8 8e 84 00 00)

(01) newObject; size: #0
(28) styleType; no compression [1]
(01) newObject; size: #0
(29) inkType; no compression [1]
(01) newObject; size: #0
(2a) transformType; no compression [1]
(05) newObject; size: #4
(83) lineType; byte compression
(00 00 7d 00               {     0.0000,     0.0000}  {   125.0000,     0.0000}
(01) newObject; size: #0
(3f) trailerType; no compression
             Total Opcodes    Total Size
New
     headerType  #       1     #       4
       lineType  #       1     #       6
      styleType  #       1     #       2
        inkType  #       1     #       2
  transformType  #       1     #       2
   fontNameType  #       1     #       8
    trailerType  #       1     #       2
Set
Default
  all shapes     #       1     #       6
  grand total    #       7     #      26
```

The numbers in parentheses are the values in the file, one byte at a time. The numbers in square brackets are the reference indices. Values in curly braces are in decimal fixed point.

Graphics objects default to referring to the last object unflattened; the line in this example refers to the simple style, ink and transform in front of it. A reference allows a shape to refer to some object other than the one immediately before it.

More Miscellany

Use the up/down arrow keys to set the scrolling speed.

Use dot '.' to represent the last displayed address.

GraphicsBug Obscura

operators: - + * / % ^ | & [@*] ~ () numbers: . 0x $ # '' strings: ""

You can do simple math expressions in GraphicsBug, and a lot of the time they'll actually work.

Numbers can be entered in hexadecimal (the default), decimal and character codes. You can explicitly enter hexadecimal by preceding it with 0x or $. You can explicitly enter decimal by preceding it with #. If there's no prefix, and the number contains a letter from a to f, then it is treated as hexadecimal. Finally, if the string is simple decimal digits, with or without a decimal point, its treated by default as hexadecimal. Selecting decimal as the integer default in the Preferences dialog changes both integer and fixed point numbers to default to decimal. The fixed point default in the Preferences dialog does nothing.

```
1*2+2*3
0x00000008 #8  #0.0001 ◊◊◊◊
```

The results follow C evaluation rules and show the result in hex, decimal fixed point and as characters. You can select whether the hexadecimal uses upper case letters or not in the Preferences dialog.

```
144.44
0x01444400 #21251072  #324.2656 ◊DD◊
```

If you don't enter a leading #, it's interpreted as hexadecimal.

```
#144.44
0x009070a3 #9466019  #144.4399 ◊◊p◊
```

Arithmetic with fixed numbers works by converting the number to a 32 long first.

```
#144.44*3
0x01b151e9 #28398057  #433.3199 ◊◊Q◊
#13.3*#12.2
0x4282f0a4 #1115877540  #17026.9400 B◊◊
```

The first example works; the second does not.

The operators available are basically the same as in Macsbug: - unary minus or binary subtraction
+ unary plus or binary addition
* unary indirection or binary multiplication
/ division
% modulo (but only makes sense with positive numbers)
^ xor, but not Pascal type postfix indirection
| or
& and
@ another way to do unary indirection
~ not
() precedence

Things that don't work

```
100/-3
```

what works instead:

```
100/(0-3)
```

Conditional operators like >, <, >=, <=, ==, != work, too.

! doesn't work consistently.

```
!4==4
0x00000000 #0  #0.0000 ◊◊◊◊   (right)
4==4
0x00000001 #1  #0.0000 ◊◊◊◊   (right)
1==1
0x00000001 #1  #0.0000 ◊◊◊◊   (right)
!1==1
0x00000001 #1  #0.0000 ◊◊◊◊   (wrong)
!(1==1)
0x00000001 #1  #0.0000 ◊◊◊◊   (also wrong)
```

Characters can be used alone or in expressions.

```
'grfx' ~'grfx'
0x67726678 #1735550584  #26482.4002 grfx  0x988D9987 #-1735550585 -#26482.4002 ◊◊◊
```

You can enter in more than one expression on the same command line.

```
1 2 3 4
0x00000001 #1  #0.0000 ◊◊◊  0x00000002 #2  #0.0000 ◊◊◊
0x00000003 #3  #0.0000 ◊◊◊  0x00000004 #4  #0.0000 ◊◊◊
```

Multiple commands can be separated by semicolons. It's the same in Macsbug.

Here's an example that works in both:

```
1;2;3;4
0x00000001 #1  #0.0000 ◊◊◊  0x00000002 #2  #0.0000 ◊◊◊
0x00000003 #3  #0.0000 ◊◊◊  0x00000004 #4  #0.0000 ◊◊◊
```

The Stuff in the Menus

The File Menu

New

You can create more than one GraphicsBug window, but if you switch back and forth between them, GraphicsBug may get confused about which window to draw into. It's best to stick to one window. GraphicsBug will successfully remember the size and placement of the window, though.

Open...

In addition to opening text files, you can open files created by FLatten, printer spool files and Portable Digital Documents.

PDD Info...

This opens a new window and generates object subtotals and totals for files created by FLatten, printer spool files and Portable Digital Documents.

Save, Save As, Save a Copy As, Revert

Save, Save As, and Save a Copy As work. Revert does not work. As we noted before, multiple windows in GraphicsBug is not very functional, so do not be cavalier about saving dumps that you care about.

Page Setup, Print

These commands do not work correctly.

Preferences

A few preferences work, but most don't. The only part of preferences that is ever worth changing is the size of the window buffer. To save a large heap dump, for instance, increase the size of the window buffer before opening the GraphicsBug window. Other changes that do work on the surface, like changing the Integer default from Hexadecimal to Decimal, goof up commands like Command-clicking on addresses. It's best to ignore Preferences.

Edit Menu

Undo never works. Cut, Copy and Paste work somewhat, but be sure to click either above or below the command line first. The most useful command in Edit is Select All; to clear out the info in a GraphicsBug window, click on the content area, execute Select All, then press delete.

Command Menu

These commands are most useful from the command line, but for completeness, they can be chosen from the menu instead. If they usually take an argument, then that argument must be selected in the content area. The most useful of the bunch is Find, which is associated with command-F. You can double click on addresses and Find them in rapid succession.

Similarly, the ERror menu item uses the current selection as the error number to look up; probably useless because the only places error numbers might show up are already translated to strings. Validate might be useful if Validate worked with an argument, but it doesn't, so it isn't. For the rest of the commands, if they work at all (and most don't) there's nothing that they do that can't be done from the command line, or in the case of DisplayMemory, option-double clicking on the address.

Heap Menu

The Heap menu has the same interface and caveats as the Command menu. Its useful contribution is the list of heaps that GraphicsBug can operate on at the bottom of the menu. If there's no Debug INIT installed, you'll get numbers instead of names.

Clients, Processes and Windows Menu

These menus are only lists of the graphics clients, Process Manager processes and windows that GraphicsBug knows about. Clients do the equivalent of a `ListClient` command; any process does a `ListProcesses` command; and the window menu attempts to bring the menu to the front. These menus can get in the way or be fooled, however; the Processes menu frequently takes over menu keys, while the Window menu doesn't do anything sensible if two windows have the same name.

Monitor Menu

The first two items, Show Fields and Show Blocks bring up windows that are constantly updated. Neither shows up in the Windows menu, nor behaves very well as windows. For instance, closing the window with the close box may cause GraphicsBug to crash. Use the menu instead. If the windows don't appear when you select them from the menu, they may have been placed behind the text window; try resizing or moving it.

Show Fields shows a constantly updated version of the current heap's header. Unfortunately, it is out of date; some fields are omitted, and others, starting with stackTop are incorrect. It's still useful for monitoring the totalFree field as the application runs.

Show Blocks shows a constantly updated graphic representation of the heap. It's another stillborn GraphicsBug idea; you can't scroll through the blocks, so if the heap is large, you'll only be able to see the first blocks. You can choose Small Blocks and see about a half of a meg. The Update Blocks choice makes the window update as the application runs. The color of the blocks show whether the blocks are direct, indirect or free. The Pattern Blocks menu item shows that when GraphicsBug was created, many of the graphics engineers at Apple were still using black and white monitors. Go figure.

If you click and drag on the blocks in the Show Blocks window, it will show you the block size, type and address.

More GraphicsBug Bugs To Watch Out For

- Sometimes negative fixed point values are off by 1.0.
- Something about the GX debug INIT and the Metrowerks debugger don't agree with each other, at least on a 68K machine. If you find yourself in Macsbug unexpectedly, try using the Macsbug DX command to disable user breaks until you can save your work and restart gracefully.
- Selecting Commands (like Validate) without running a GX application will result in a bus error. Running GraphicsBug without GX installed may also crash. Also, many commands and displays are far less descriptive without the debugging INIT installed. There's rarely a reason not to run the debugging INIT while developing a GX application.

References

Developer CD Series: Mac OS SDK Edition: Development Kits (Disc 2): QuickDraw GX: Programming Stuff: GX Libraries:

Inside Macintosh: QuickDraw GX Objects

Inside Macintosh: QuickDraw GX Environment and Utilities

Back to top

Downloadables

Acrobat version of this Note (444K). Download

Back to top