Apple IIgs Assembly Programming Crash Course

Dagen Brock

dagenbrock@gmail.com

KansasFest 2013

Sources Online:
https://github.com/digarok/gslib

In The Beginning:

The following diagram shows a simple program that can be compiled in Merlin 16 and run from a standard AppleSoft basic prompt. Since there's no **ORG** statement at the top, Merlin will compile it at $8000, but it really doesn't matter. This program is a single instruction, **RTS**. It can be loaded to, and run, from anywhere!

```
*****************************************
* Quit                                  *
*****************************************

                rts             ; return from wherever we were called
```

If you want to try it,

- launch Merlin and Load the "quit" source file (you don't need to type the ".s"), this will leave you in the full-screen editor
- hit OpenApple-A to assemble the code, then hit escape to go back to the main menu
- save the Object file as "quit", then quit out of Merlin and launch basic
- assuming you are in the right directory in BASIC, you can type "BRUN QUIT"
- you could also "BLOAD QUIT" then "CALL 32768" (that's hex $8000)
- or "BLOAD QUIT" then "CALL -151" to enter monitor and "8000g" to run it.

Prodos 8:

If you were to write the same thing, in ProDOS8, a program that just exits immediate, then you would have to do things a bit differently. ProDOS8 introduces the concept of calling a Quit routine. In this case, you can try the program yourself by just Loading "quit8" then hitting OpenApple-A to assemble. Thanks to the **DSK** and **TYP** commands (actually compiler directives), it will automatically write out a P8 system file that you can launch from GSOS or any ProDOS launcher.

```
*****************************************
* Quit8                                 *
*                                       *
*  Dagen Brock <dagenbrock@gmail.com>   *
*  2013-06-24                           *
*****************************************

                org $2000       ; start at $2000 (all ProDOS8 system files)
                dsk quit8.system ; tell compiler what name for output file
                typ $ff         ; set P8 type ($ff = "SYS") for output file

MLI             equ $bf00



Quit            jsr MLI         ; first actual command, call ProDOS vector
                dfb $65         ; with "quit" request ($65)
                da QuitParm
                bcs Error
                brk $00         ; shouldn't ever  here!

QuitParm        dfb 4           ; number of parameters
                dfb 0           ; standard quit type
                da $0000        ; not needed when using standard quit
                dfb 0           ; not used
                da $0000        ; not used


Error           brk $00         ; shouldn't be here either
```

GSOS/ProDOS16:

Similar to ProDOS 8, GSOS expects you to call a quit routine. The main new concepts here are that we are telling the compiler to create relocatable code that can be loaded anywhere in memory, via the **REL** command. Since our programs can be loaded anywhere, we always want to start by pushing our program bank register and pulling it back as our data bank. GSOS doesn't guarantee that it will point the data bank to where it loads your program because it doesn't know where you want to access data banks!

```
*****************************************
* Quit16                                *
*****************************************

                rel             ; compile as relocatable code
                dsk Quit16.l    ; Save Name

                phk             ; Set Data Bank to Program Bank
                plb             ; Always do this first!

                jsl $E100A8     ; Prodos 16 entry point
                da $29          ; Quit code
                adrl QuitParm   ; address of parameter table
                bcs Error       ; never taken

Error           brk             ; should never get here

QuitParm        adrl $0000      ; pointer to pathname (not used here)
                da $00          ; quit type (absolute quit)
```

If you want to try it,

- hit OpenApple-O to open the "Command" window and type "link", this will assemble and link your file, and create an executable System16 binary called "Quit16"
- you can launch that from the GSOS desktop
- or, from Merlin's main menu, you can hit Disk Command and type "-quit16" to launch directly

SHR (Super Hi-Res) Graphics Mode:

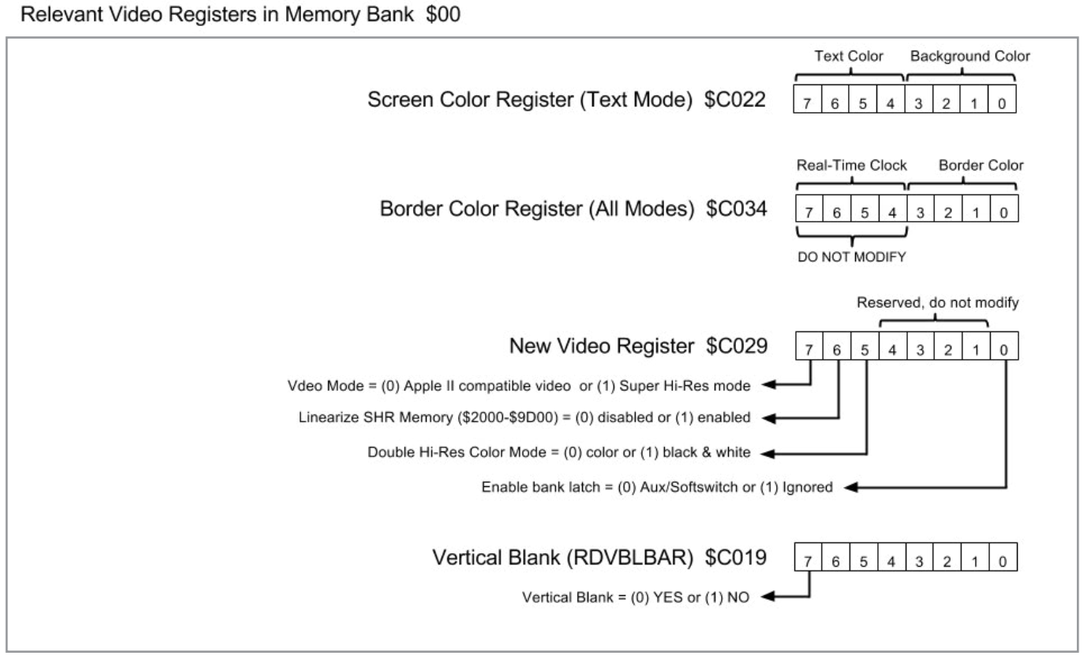
There are actually two SHR modes on the Apple IIgs, 320x200 and 640x200. We will only discuss the 320 mode which supports 16 colors per palette, with up to 16 palettes for a total of 256 colors on screen. You can actually go higher than this, but that requires writing precisely timed code to swap palettes and control bytes while the screen is updating.

The IIgs also lets you set the text, background and border color of the "text" screen. I mention this, because it's common to set the border color to black or some other color during games or graphical displays.

The IIgs display is mapped to bank $E1, starting at memory location $2000. It can also be shadowed into bank $01. You will quickly see that much graphics code regularly references $E12000 or $012000.

The IIgs also has a video blanking register that you can read and syncronize your code with to create flicker free effects.

Unfortunately, there's no built-in sprite hardware and the screen memory is locked at a dismally slow 1MHz, but luckily it's quite easy to program for.



Turning on SHR mode:

Here's a simple, but common, routine for turning on the SHR graphics mode. Note, we switch to 8-bit to set flags only on that byte at bank $00, location $C029.

Comparing it to the table above, you can see we set bits:
7 = SHR mode
6 = Linearize the SHR memory at locations $2000-9CFF
1 = Enable Bank Latch

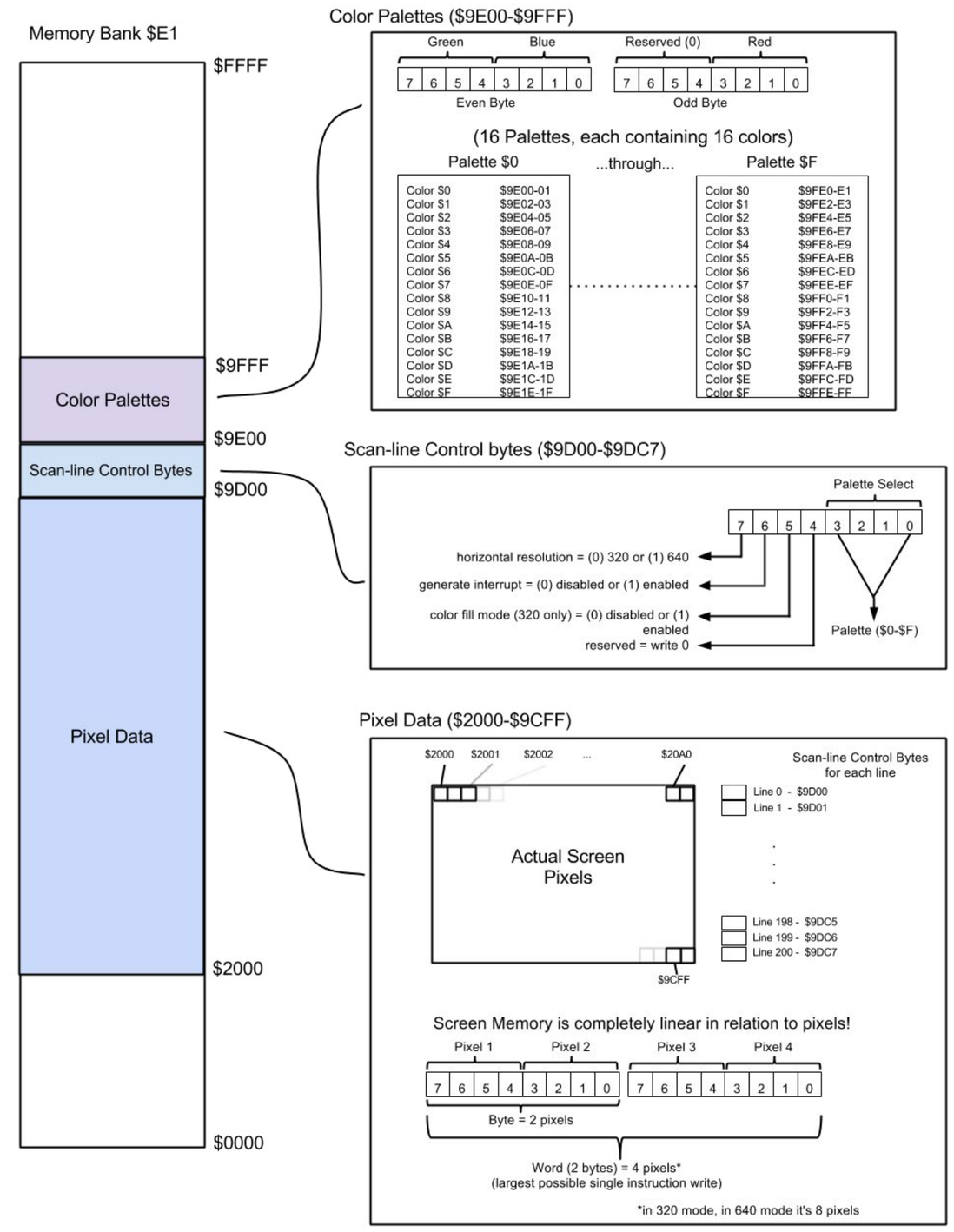
```
        ***************************************
        * Turn on SHR mode                    *
        ***************************************
        GraphicsOn      sep #$30        ;8-bit mode
                        lda #$C1        ;%1100 0001
                        stal $00C029    ;Turn on SHR mode
                        rep #$30        ;back to 16-bit mode
```

SHR Memory Map:

The next page illustrates the Apple IIgs memory map.

Refer to the online sources to see:

shr1.s	SHR1, Shows how to turn on SHR Graphics Mode and clear screen
shr2.s	SHR2, Shows how to write palettes, clear screen to color, set scan-line control bytes



DAGEN'S MEGA SUPER HI-RES GRAPHICS REFERENCE PAGE!

Toolbox:

The Apple IIgs contains an entire ecosystem of routines that are included in the firmware. It's called the Toolbox and later versions of ProDOS16 and GSOS also update those tools by loading new tools in RAM. (You can even write your own tools!)

We won't concern ourself with a majority of tools, like those used when writing desktop applications. It's enough to simply know about the "big five".

Tool Locator: Takes care of loading and dispatching tools calls so you don't have to think about it!
Memory Manager: Allocates memory. You must use this to request memory for your application
Miscellaneous Tool Set: Random system-level routines. We'll use the "UnPackBytes" toolcall
QuickDraw II: Handles drawing routines for desktop software. We will NOT be using this!!!
Event Manager: Traps all events (mouse/keyboard) and provides a convenient queue for processing events in your application

How to make a tool call:

Tools are meant to run in full native 65816 mode with all 16 bit registers. You call them using a common vector, kind of like calling ProDOS 8 calls. The entry point for the Toolbox routines is $E1/0000.

Here's how to call a tool:

- Push zeros (or anything) on the stack to make room for any results, depending on whether the tool you are calling returns anything
- Load the 2-byte tool call number into X. e.g. "`LDX #$0201`" calls tool $01, command $02
- Do a `JSL $E10000` to call the Toolbox dispatcher
- Pull any results off the stack, again depending on whether the tool you are calling returns anything
- Check the carry flag for an error. Carry will be set if an error occured during the last tool call and the Accumulator will hold the error code. Otherwise, Carry and Accumulator will be zero

```
****************************************
* Call TLStartUP                       *
* - there are no parameters to pass or *
*   results to pull off of the stack   *
****************************************
               ldx #$0201      ;the code for TLStartup
               jsl $E10000     ;call the Toolbox vector
```

Here's an example that passes variables and returns a result. This particular piece of code requests a block of memory from the memory manager.

```
****************************************
* Call NewHandle                       *
* - PushLong and PushWord and ToolCall *
*   are Merlin macros that are         *
*   INVALUABLE in Toolbox programming  *
****************************************
               PushLong #$0000 ;space for result
               PushLong #$2000 ;size of block needed
               PushWord MyID   ;your applications userID
               PushWord #$0000 ;attribute byte for your block
               PushLong #$0000 ;address requests (or none, in this case)
               ToolCall $0902  ;NewHandle
               PullLong MyHandle ;don't forget to pull your result off the stack
               bcc :noErr      ;error handling would follow
```

Memory Manager:

You use the memory manager to allocate memory for your program. You may have some memory "baked-in" to your program, like variables and tables generated at compile time. But if you want to request a block of memory, perhaps to load a file and store it somewhere, you will use the NewHandle call from the Memory Manager.

When you first start your program, you will commonly start the Tool Locator, then you will start the Memory Manager and get a UserID that you will use for subsequent memory requests.

```
****************************************
* Typical tool startup                 *
****************************************
               _TLStartUp      ;start Tool Locator
               pha
               _MMStartUp      ;start Memory Manager
               bcs :Error      ; should never happen
               pla
               sta MasterId    ;master handle references the memory allocated to us
               ora #$0100      ;set auxID = $01  (valid values $01-0f)
               sta UserId      ;any memory we request must use our own id
```

Requesting a block of memory:

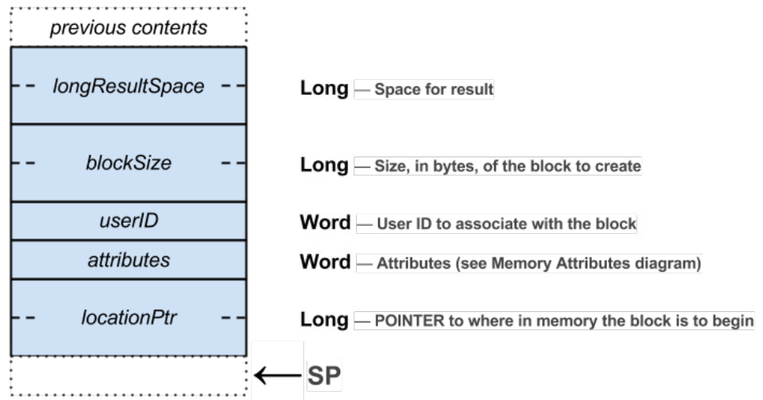
You request memory from the memory manager by the use of the NewHandle call. You assign it to your UserID and tell the memory manager what attributes it has, like whether or not it's relocateable.

Memory Manager Tool Set ($02)

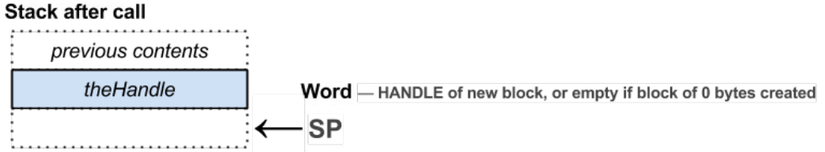
Tool Call $0902 NewHandle

Parameters

Stack before call



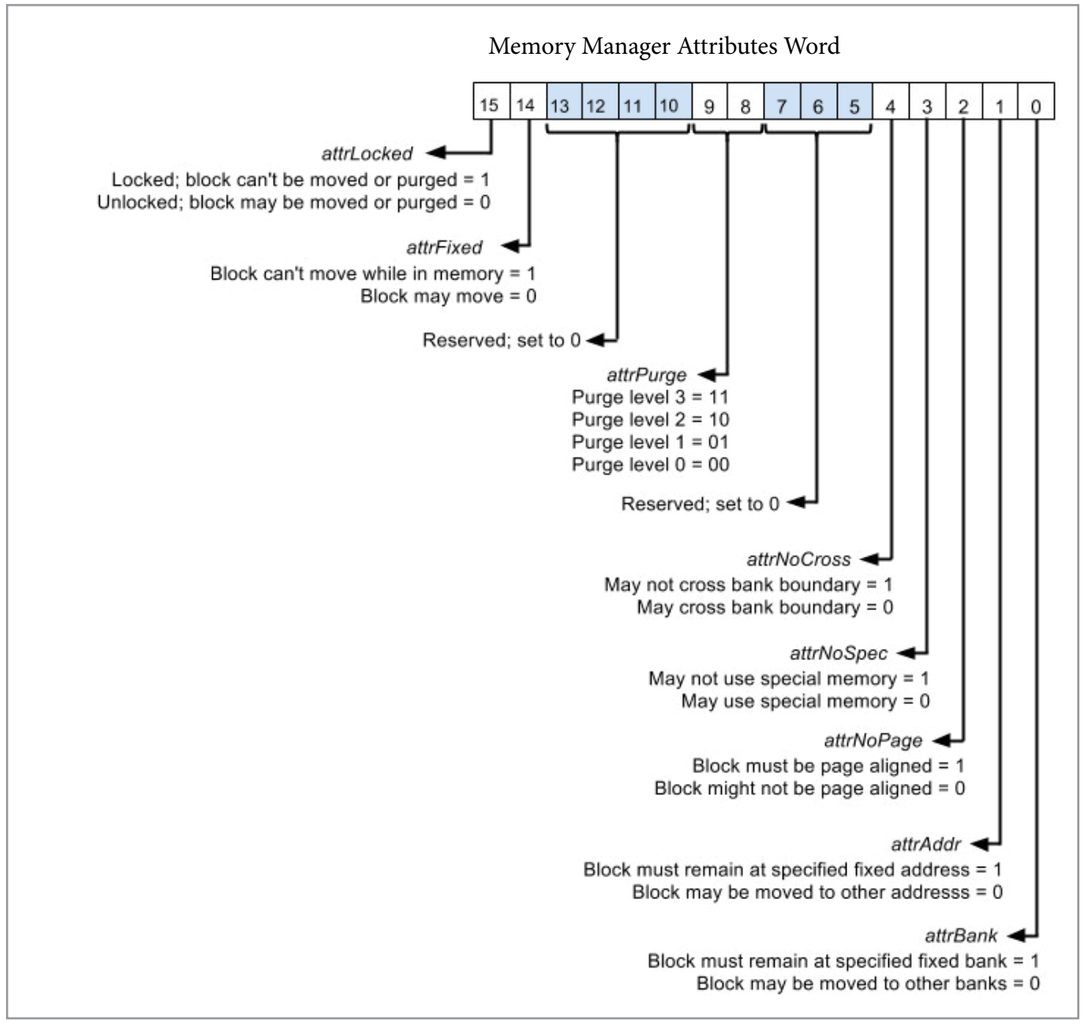
Stack after call



Memory Manager Attributes:

When requestiong memory, you pass in an attribute that specifies what kind of restrictions you'd like on your memory block. The common/main ones for programmers is to make the block **locked** so you can dereference the handle to get your pointer and, from then on, just work on that block of memory directly, as opposed to always checking to make sure it hasn't moved.

You can also specify an address, using the locationPtr parameter, but if you want to do that you must also set the attributes for fixed address and/or fixed bank.



Refer to the online sources to see:

shrloadimg1.s	Shows how to start tools, request memory, load and unpack picture
shrloadimg2.s	Same as above, but with some fade/palette routines and memory shadowing

PackBytes/UnPackBytes:

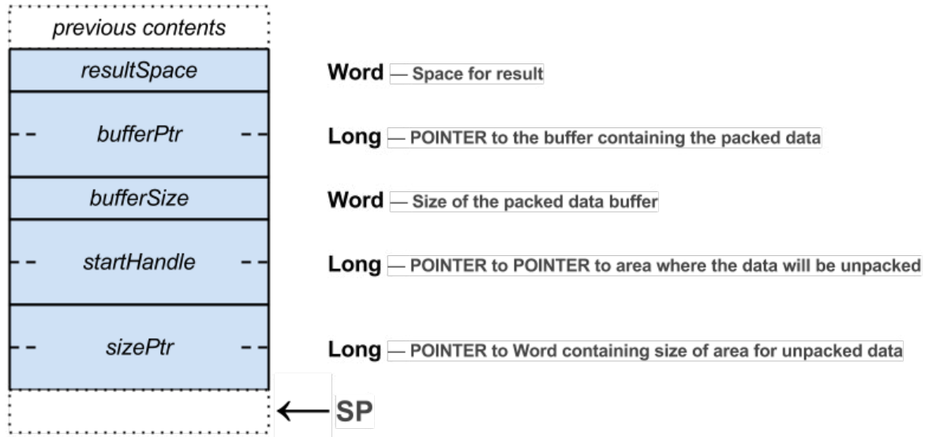
I want to mention the UnPackBytes routine, because I use that in examples to load SHR "packbytes" images. These images are smaller, so you can load them faster and they take up less memory. It's worth using PackBytes images, even if you don't understand how it works. Many Apple IIgs Paint programs can save in the PackBytes format.

Miscellaneous Tool Set ($03)

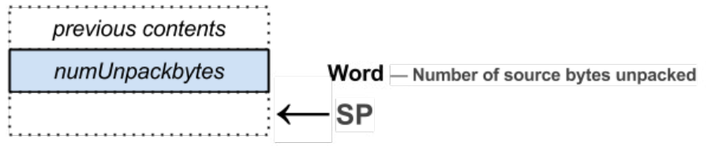
Tool Call $2703 UnPackBytes

Parameters

Stack before call



Stack after call



Refer to the online sources to see:

shrloadimg1.s	Shows how to start tools, request memory, load and unpack picture
shrloadimg2.s	Same as above, but with some fade/palette routines and memory shadowing

Sprites:

As mentioned, there is no sprite hardware on the IIgs. You simply copy pixel data to the screen memory. This means you are on your own to write sprite routines.

Generally, you can just create tables of sprites and copy them over

```
BALL            hex 00FFFF00
                hex 0FFFFFF0
                hex 0FFFFFF0
                hex 0FFFFFF0
                hex 0FFFFFF0
                hex 00FFFF00

DrawBallAtX     lda Ball
                stal $E12000,x
                lda Ball+2
                stal $E12002,x
                lda Ball+4
                stal $E120A0,x          ;next line, so add 160 (hex $A0)
          ... etc
```

The problem with this method is that it draws over everything. What if we want to see some background surround our ball? With the above code it will have a box arount it.

The solution is to use masking. We load the screen data, mask off the area to draw our pixels using AND, then combine our sprite data with the screen data using OR. Here's a visual example:

Load

```
SCREEN
hex 33333333
hex 33333333
hex 33333333
hex 33333333
hex 33333333
hex 33333333
```

*get screen data

Logical AND

```
BALLMASK
hex FF0000FF
hex F000000F
hex F000000F
hex F000000F
hex F000000F
hex FF0000FF
```

*mask off hole area

Result

```
MASKRESULT
hex 33000033
hex 30000003
hex 30000003
hex 30000003
hex 30000003
hex 33000033
```

* we've created a hole

Logical OR

```
BALL
hex 00FFFF00
hex 0FFFFFF0
hex 0FFFFFF0
hex 0FFFFFF0
hex 0FFFFFF0
hex 00FFFF00
```

* now apply sprite data

Save Result

```
SCREENRESULT
hex 33FFFF33
hex 3FFFFFF3
hex 3FFFFFF3
hex 3FFFFFF3
hex 3FFFFFF3
hex 33FFFF33
```

*result is masked sprite

There are a lot of ways to optimize this, like only masking bytes or words where there is a transparent area. For more information on this, and a great tool for compiling optimized sprites, I recommend you look at "Mr. Sprite" by Brutal Deluxe. He does a much more in-depth explanation of the technical concepts behind drawing to the IIgs screen and has nice color versions of the above tables that really make things clear.

http://www.brutaldeluxe.fr/products/crossdevtools/mrsprite/

Keep in mind, this isn't specific to only the Apple IIgs. This is a common paradigm for systems that have some sort of memory mapped display area and no specialized sprite routines.

Input (TBA):

You can use your standard Apple II keyboard routines, as we have here, or use the event manager. I'll discuss the latter in an online update, as well as add some mouse routines.

Sound (TBA):

You can program the Ensoniq DOC directly, or using Tools. There are built in sound tools, as well as some really great third-party tools for playing music. I'll try to cover these in future online updates.

THANKS FOR COMING TO KANSASFEST!