Apple IIGS
#65: Control-^ is Harder Than It Looks

Written by: Dave Lyons September 1989

This Technical Note describes a problem using Control-^ to change the text cursor with programs that use GETLN.

On the Apple IIGS, typing Control-^ changes the cursor to the next character typed. This feature works properly from the keyboard, but there is a problem when programs print the control sequence. Try entering the following from AppleSoft to demonstrate this problem:

```
NEW
PRINT CHR$(30);"_"
```

It changes the cursor into a blinking underscore, as expected. But now enter the following:

```
12345 HOME
LIST
```

You should see `2345 HOME`, which demonstrates that the first character is ignored. This is a problem with GETLN, which AppleSoft uses to read each line of input. Even if your program does not use this routine, you should be aware of this problem since it will occur the next time another program uses GETLN.

Since changing the cursor works fine when done from the keyboard, the way to work around this problem is to have your program simulate the appropriate keypresses for GETLN.

```
301: CLD            ; required by BASIC.SYSTEM
302: STA ($28),Y    ; remove cursor if present
304: LDY $0300      ; get index into simulated-keys list
307: LDA $310,Y     ; get a simulated keypress
30A: INC $0300      ; point to the next key for next time
30B: RTS            ; return the key to GETLN

310: 9E DF 8D       ; Ctrl-^, underscore, return

100 POKE 768,0 : PRINT CHR$(4);"IN#A$301" : REM Start getting simulated keys
110 INPUT "";A$
120 PRINT CHR$(4);"IN#0" : REM Get real keys again
```

From an assembly-language program, the equivalent of `IN#A$301` is storing $01 and $03 in locations $38 and $39, while the equivalent of `INPUT` is `JSR $FD6A` (GETLN). (Store a harmless prompt character, like $80, into location $33 first.)

Further Reference

- *Apple IIGS Firmware Reference*, p. 77

Apple IIGS
#66: ExpressLoad Philosophy

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage September 1989

This Technical Note discusses the ExpressLoad feature and how it relates to the standard Loader and your application.

Changes since September 1990: Clarified some changes now that ExpressLoad and the System Loader are combined to be "Loader 4.0" in System Software 6.0. Completely removed the note about not calling `Close(0)` since it's not relevant.

Speedy the Loader Helper

ExpressLoad is a GS/OS feature which is usually present with System Software 5.0 (if the `ExpressLoad` file is present and there's more than 512K of RAM), and always on System Software 5.0.4 and later. In fact, ExpressLoad is no longer a separate file in System Software 6.0; it's included in the System Loader version 4.0. Even though ExpressLoad is part of the Loader, we refer to its functionality separately to distinguish how the Loader takes special advantage of "expressed" files.

ExpressLoad operates on Object Module Format (OMF) files which have been "expressed," using either the APW tool Express (or it's MPW counterpart, ExpressIIGS) or created that way by a linker. Expressed files contain a dynamic data segment named either `ExpressLoad` or `~ExpressLoad` at the beginning of the file. (Current versions of Express and ExpressIIGS create `~ExpressLoad` segments, which is the preferred naming convention; older versions created `ExpressLoad` segments, and should be re-Expressed for future compatibility.) This segment contains information which allows the Loader to load these files more quickly, including such things as file offsets to segment headers, mappings of old segment numbers to new segment numbers (these files may have their segments rearranged for optimal performance), and file offsets to relocation dictionaries.

Two Loader Components, Two Missions, One Function

The System Loader's function is to interpret OMF. It takes files on disk (or in memory) and transforms them from load files into relocated 65816 code. It does this very well, but in a very straightforward way. For example, when the System Loader sees the instruction to right-shift a value n times, it loads a register with the value and performs a right-shift n times.

ExpressLoad has a different mission. It relies upon the rest of the System Loader to handle OMF in a straightforward fashion so it can concentrate upon handling the most common OMF cases in the fastest possible way. For example, when asked for a specific segment in a load file, the System Loader "walks" the OMF until it finds the desired segment. ExpressLoad, however, goes directly to the desired segment since an Expressed file contains precalculated offsets to each segment in the `ExpressLoad` segment.

Since ExpressLoad focuses on the common things performed by the majority of applications, it may not support those applications which rely upon certain features of OMF or the System Loader. In these cases, the System Loader loads the file as is expected.

ExpressLoad always gets first crack at loading a file, and if it is an Expressed file that ExpressLoad can handle, it loads it. If the file is not an Expressed file, the regular System Loader loads it instead. ExpressLoad also gets first shot at other loader calls.

Because an Expressed file is a standard OMF file with an additional segment, Expressed files are almost fully compatible with the System Loader (although it cannot load them any faster than before). Refer the following section for potential problems.

Working With ExpressLoad

As ExpressLoad is intimate in its relationship with the System Loader, most applications work seamlessly with it; however, there are some potential problems about which you should be aware.

- Don't mix Expressed files and normal OMF files with the same user ID. For example, if your application uses `InitialLoad` with a separate file, make sure that if it and your main application share the same user ID that they are both either Expressed files or normal OMF files.

- Don't use a user ID of zero. In the past, use of zero told the System Loader to use the current user ID; however, now both the System Loader and ExpressLoad have a current user ID. Be specific about user IDs when loading. This is fixed in 6.0, but is still a good thing to avoid for compatibility with System Software 5.0 through 5.0.4.

- Avoid loading and unloading segments by number. Since Expressed files may have their segments rearranged, if an Expressed file is loaded by the System Loader, references to segments by number may be incorrect.

- Avoid using `GetLoadSegInfo` before System Software 6.0. This call returns System Loader data structures which are not supported by ExpressLoad previous to 6.0. In System Software 6.0 and later, the combined Loaders return correct information for `GetLoadSegInfo` regardless of whether the load file is expressed or not.

- Don't try to load segments in files which have not been loaded with the call `InitialLoad`. This process was never a very good idea, and it is now apt to cause problems.

- Don't have segments that link to other files. ExpressLoad does not support this type of link.

Further Reference

- *GS/OS Reference*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#67: LaserWriter Font Mapping

Revised by: Matt Deatherage May 1992
Written by: Suki Lee & Jim Luther September 1989

This Technical Note discusses the methods used by the Apple IIGS Print Manager to map IIGS fonts to the PostScript® fonts available with an Apple LaserWriter printer.
Changes since November 1989: Corrected some typographical errors and added Carta and Sonata, two fonts the LaserWriter driver knows about but aren't built into any LaserWriter.

Version 2.2 and earlier of the Apple IIGS LaserWriter driver depend solely upon font family numbers as unique font identifiers. There is a table built into the driver which maps the known font family numbers to the built-in LaserWriter family fonts. Any fonts which are not built-in are created in the printer from its bitmap font strike. Under this implementation, all font family numbers not known at the time the driver was written print using bitmap fonts. This driver knows nothing of any other fonts which may reside in the printer.

There have been many requests for the driver to take advantage of other available PostScript fonts to get high quality output from the LaserWriter. PostScript fonts from Adobe's font library, or from other PostScript font manufacturers, can be downloaded to the printer from a Macintosh and remain in the printer for use until power off. Currently there is no means to download a PostScript font with an Apple IIGS.

The Apple IIGS LaserWriter driver version 3.0 and later makes use of most resident PostScript fonts in the LaserWriter when requested. If the font is not available, then the bitmap font is used. The driver queries the printer at the start of a job for the font directory listing. The listing consists of names of all the fonts in the printer, built-in or downloaded. This information is kept locally for look up using the name of the requested font.

Issues

All Apple IIGS fonts contain a family name and a family number. The Apple IIGS currently identifies fonts using the family number; however, this identification method may change in the future, due to the complexity of tracking unique matches between font family names and font family numbers.

PostScript identifies its fonts by name (case sensitive) and knows nothing of any font family numbering system, Macintosh or Apple IIGS, which might be attached to a particular font. Most

PostScript font families include plain, bold, italic and bold italic fonts. Some fonts families may also have serif and sans serif fonts or fonts of different weights (line thickness). These fonts are generally named by adding a style suffix to the base family name. Unfortunately, there is no uniform method for naming fonts, since most fonts were named by their designers and many of the names have historical significance.

The three examples shown in Table 1 show three variations of the plain font, two variations of the bold style, three variations of the italic style, and three variations of the bold italic style. There are others such as ZapfChancery-MediumItalic, Korinna-KursivRegular, and LetterGothic-Slanted which all denote the italic style of the respective font family.

Style	Font names		
plain	Helvetica	Times-Roman	AvantGarde-Book
bold	Helvetica-Bold	Times-Bold	AvantGarde-Demi
italic	Helvetica-Oblique	Times-Italic	AvantGarde-BookOblique
bold italic	Helvetica-BoldOblique	Times-BoldItalic	AvantGarde-DemiOblique

Table 1–Example Font Names

The Macintosh LaserWriter driver uses a mapping scheme to compose a full PostScript font name. It relies on the Font Family Definition Record `'FOND'` resource to provide a style mapping table containing the appropriate suffixes.

There are no similar resources on the Apple IIGS, which means the Apple IIGS LaserWriter driver has no way to match PostScript fonts to Apple IIGS fonts. Instead, the Apple IIGS LaserWriter driver adopts the following approach. The driver has full knowledge of all LaserWriter family built-in fonts (see Table 2 for a list of these built-in fonts) plus Carta and Sonata (two graphical fonts used in map and music programs) and uses the correct name for all style variations of the fonts. For all other fonts, the driver uses a standard set of suffixes for the style modifications. These suffixes are -Bold, -Italic, and -BoldItalic. The appropriate suffix is appended to the family name of the font, and this name is used to search the font directory table obtained from querying the printer. If a match is found, the document is printed using the corresponding PostScript font. If no match is found, then the driver tries to find the plain form of the font and creates the style modification in PostScript. A bitmap of the font is downloaded to the printer if these two searches fail.

If you are shipping your application with the intention of taking advantage of PostScript fonts when printing to a LaserWriter, please be sure to provide an Apple IIGS font whose family name is identical to the PostScript font family name.

All LaserWriters	LaserWriter Plus and LaserWriter II	
Courier	AvantGarde	Palatino
Carta	Bookman	Symbol
Helvetica	Courier	Times
Sonata	Helvetica	ZapfChancery
Symbol	Helvetica-Narrow	ZapfDingbats
Times	NewCenturySchlbk	

Table 2–Built-in LaserWriter Fonts

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1 & 2
- *Apple LaserWriter Reference*

Apple IIGS
#68: Tips for I/O Expansion Slot Card Design

Written by: Rob Moore & Jim Luther September 1989

This Technical Note points out several potential problem areas developers should know about when designing I/O expansion slot cards for the Apple IIGS.

This Note is written for experienced design engineers. It is not intended to be a tutorial on Apple IIGS I/O expansion card design techniques, but rather to point out possible problem areas and pitfalls to help developers produce successful and reliable expansion cards.

The 65C816 PH2 Clock versus the Expansion Slot PH0 Clock

It is important to understand the timing of the 65C816 Phase 2 clock (PH2) on the IIGS, because several of the expansion slot signals are actually related to the PH2 clock timing, rather than the 1 MHz Phase 0 clock (PH0) available at the expansion slots. Unlike the Apple IIe, the PH2 clock at the CPU is not the same as the PH0 clock found at the expansion slots. The PH2 clock runs at a variety of periods, depending on whether the CPU is doing a normal 350 nanosecond 2.8 MHz cycle, a extended 700 nanosecond RAM refresh cycle, an isolated slow cycle, or consecutive 980 nanosecond 1.024 MHz slow cycles. During isolated slow cycles, or the first of a series of consecutive slow cycles, the fast side of the system must wait to synchronize with the 1 MHz side of the system. This synchronization results in an average cycle time of about 1.5 microseconds.

Cycle Type	Low	High	Period
Normal 2.8-MHz cycle	140ns	210ns	350ns
Refresh extended cycle	140ns	560ns	700ns
Isolated 1-MHz cycle	140ns typ.	1.33μs avg.	≈1.5μs
Consecutive 1-MHz cycles	140ns	840(980)ns	980ns

Table 1–PH2 Clock Times

The Mega II Select Signal

On the Apple IIGS, the Mega II select signal (/M2SEL) is used as the enable to the slower, 1 MHz side of the system. It goes active (low) whenever the 1 MHz side RAM or I/O areas are accessed. Accesses that cause /M2SEL to be asserted include shadowed video writes, any

accesses to internal I/O or expansion card slots, and accesses to banks $E0 and $E1. Accesses to any expansion card ROM areas that are set to Internal ROM with the Slot register do not assert the /M2SEL signal and run at the 2.8 MHz speed rather than the normal 1 MHz expansion card speed. Also, accesses to the Shadow register ($C035), CYA register ($C036), or DMA bank register ($C037), and reads from the Slot register ($C02D) or State Register ($C068) run at full speed since they are done wholly on the fast side of the system.

/M2SEL can be viewed as an extension of the address bus on the expansion slots. When it is active, it indicates that the CPU is running synchronized with the 1 MHz side of the system and the address on the address lines is a valid Apple II address in the 128K main or auxiliary memory space.

The Mega II Bank 0 Signal

The Mega II bank 0 signal (M2B0) provides the least significant bit of the CPU or DMA bank address to the 1 MHz side of the system. It is normally tri-stated and goes active for 140 nanoseconds, starting 140 nanoseconds after the PH0 clock falls. During the 140 nanosecond active period, M2B0 will be high whenever the CPU is accessing bank $E1 (with the exceptions noted previously) or doing a shadowed video write or I/O access in bank $01. Note that M2B0 does not reflect the state of the RAMRD, RAMWRT, ALTZP, 80STORE, or PAGE2 soft switches that allow access to the auxiliary 64K through bank $00. It only indicates accesses to bank $E1 or shadowed accesses through bank $01.

It is generally safe to latch the state of M2B0 by using the falling edge of the Q3 clock. Even though M2B0 will be tri-stated at the about the same time as Q3 falls, the turn-off and float time on M2B0 will generally provide sufficient hold time provided that there is not more than 1 LS TTL load on M2B0.

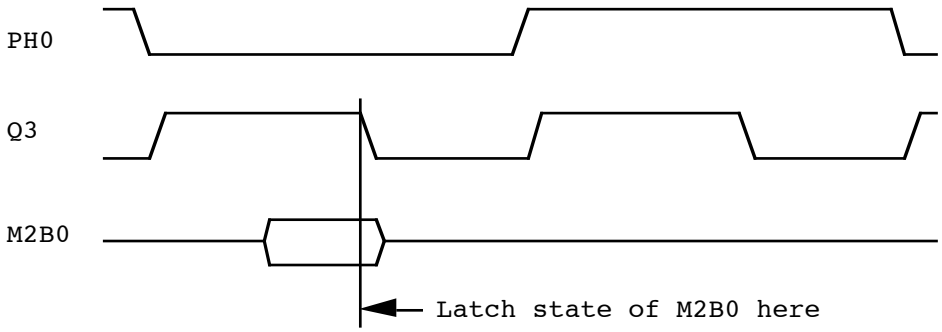

Figure 1–When to Latch State of M2B0

The Apple Video Overlay card uses M2B0 to detect writes to main and auxiliary RAM so that it can capture writes to the Apple IIGS video display buffers into its on-card display buffer. M2B0 is designed for this sort of thing and isn't of much use in most other applications. Note that M2B0 is only available on slot 3.

Using the Ready Signal

The Ready (RDY) input to the 65C816 is used to prevent a CPU cycle from completing until the expansion card has accepted the data output or has its input data available.

When the RDY input to a 65C02 or 6502 is held low, the processor continues to output the same address until RDY is released and the CPU completes the current cycle.

In the Apple IIGS, the 65C816 samples the RDY input when the PH2 clock goes low, and if RDY is low, the current CPU cycle does not complete and the address continues to be emitted. However, the bank address is not emitted while the clock is low if RDY is held low. To deal with this situation, the FPI (Fast Processor Interface) custom IC in the Apple IIGS uses a transparent latch to capture the bank address from the CPU. The latch is transparent while the PH2 clock is low and holds the bank address while the PH2 clock is high. If RDY is low, the CPU emits an invalid bank address, so the FPI holds the latch closed while RDY is low. This action is normally completely transparent to cards in the Apple IIGS expansion slots, but if an expansion card asserts RDY while the PH2 clock is low, it is likely to cause the FPI to latch an invalid bank address, because the latch could close before the bank address from the CPU is available on the data lines.

To avoid unpredictable results, RDY should only be asserted or deasserted when /M2SEL is low and when PH0 is high, or when /DEVSEL, /IOSEL or /IOSTRB are active. When /M2SEL, /DEVSEL, /IOSEL or /IOSTRB are active, you are guaranteed that the 65C816 is running at 1 MHz and is properly synchronized to the 1 MHz side of the system. RDY should be stable at least 60 nanoseconds before the falling edge of PH0 to allow for about a 25 nanosecond skew between the PH0 slot clock and the PH2 CPU clock. Figure 2 shows where it is safe to assert or deassert RDY. Limiting changes to RDY to the time when PH0 is high guarantees that it does not change while the CPU is outputting the bank address.

The RDY line should be driven with an open-collector driver.

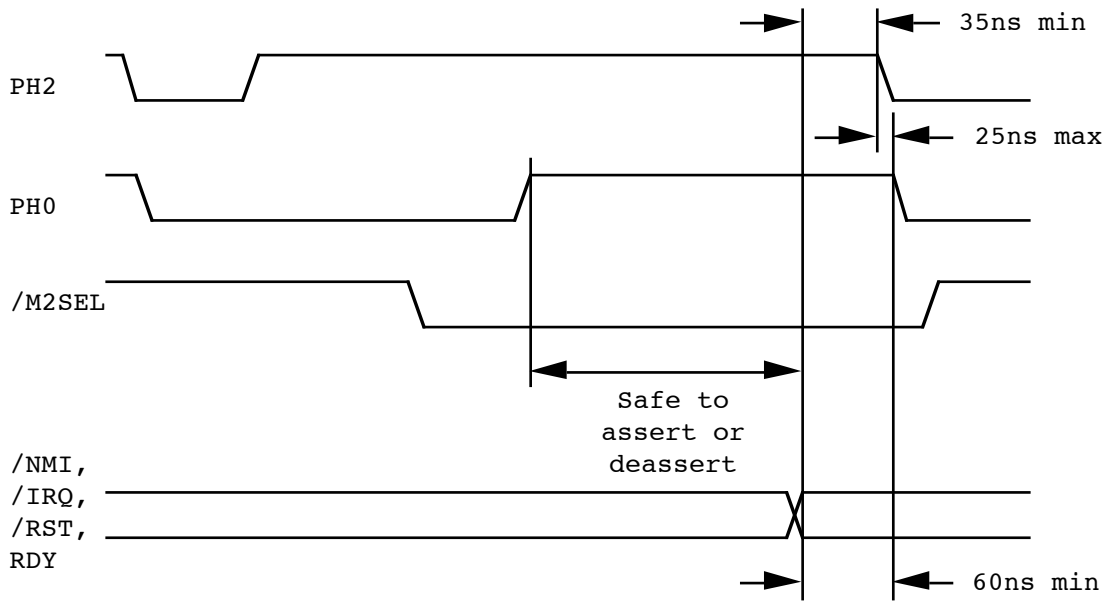

Figure 2–Control Signal Setup Time

Interrupt Request, Non-Maskable Interrupt, and Reset

The Interrupt Request (/IRQ), the Non-Maskable Interrupt (/NMI) and the Reset (/RST) signals are all interrupt lines that are sampled by the CPU when the PH2 clock falls. If they are valid 30 nanoseconds before the PH2 clock falls, they are recognized on the following cycle. If this setup time is not met, they may not be recognized until the second following cycle . Since there can be up to a 25 nanosecond skew between the PH0 and PH2 clocks, these signals should be valid 60 nanoseconds before PH0 falls if they are to be recognized on the following cycle. Figure 2 shows the correct setup time for these signals.

All three signals are all active-low and must be driven with open-collector drivers.

Note: Interrupt vectors are always pulled from ROM regardless of whether or not the language card soft-switches have ROM enabled, providing that the I/O shadowing for banks $00/01 is enabled—which it always is when running Apple IIGS or Apple II system software.

Direct Memory Access

The Direct Memory Access (/DMA) signal is used to temporarily halt the CPU and allow expansion cards direct access to the system RAM to transfer data at high speeds. Since the 65C816 is fully static while the PH2 clock is high (unlike the 6502), /DMA may be asserted for as long as necessary on the Apple IIGS.

The /DMA signal should be asserted and deasserted within the 100 nanosecond period after PH0 falls, and the DMA address should be emitted by the expansion card about 30 nanoseconds later. In any case, the address should be stable on the address bus no later than 120 nanoseconds after PH0 falls. This guarantees that there is enough time for the address to be decoded and for /M2SEL and M2B0 to be asserted by the FPI chip if the DMA transfer is to the 1 MHz side of the system. The bank address must be stored in the DMA bank register at location $C037 before using DMA.

/DMA is a active-low signal and should be driven with an open-collector driver. The Apple IIGS provides a pullup for /DMA, but since the pullup is a fairly high value, it is a good idea for an expansion card that has asserted /DMA to momentarily pull it high for a few nanoseconds when deasserting it.

Note that there is a minor hardware bug in the Apple IIGS that could cause problems for developers who are unaware of it. If the CPU is currently pulling an interrupt vector when the /DMA signal is asserted, and if the DMA address is accessing the language card ($D000-$FFFF) space in a bank of memory where I/O and language card emulation is enabled (normally banks $00, $01, $E0 and $E1), DMA reads access ROM rather than RAM. This happens because the CPU's Vector Pull (VP) signal is active while the DMA cycle is active. Since most expansion cards that use DMA are also associated with some corresponding firmware or software driver, it's a good idea to disable interrupts prior to doing the DMA transfer, then re-enable interrupts as soon as possible after the transfer is complete. If interrupts are off too long, AppleTalk shuts down any connections to file servers because the system does not respond to AppleTalk "tickle" transactions while interrupts are disabled.

We recommend that the DMA be done with the Apple IIGS running at 1 MHz. If DMA is started during a 1 MHz cycle (/M2SEL asserted), the system continues to run slow while the /DMA signal is active.

Avoiding "Bus Fights"

The data bus on the Apple IIGS (and Apple IIe) expansion slots is a multiplexed bus that is used to carry both CPU and video display data. While PH0 is low, the bus is used to transfer data from the system RAM to the video display circuitry. When PH0 is high, the bus is available for CPU data transfers. To avoid potential (or actual) bus fights, it is helpful to avoid driving read data from an expansion card onto the bus immediately after PH0 rises. Since the video read data is driven out onto the expansion slots, and expansion card read data is driven in from the slots, it takes a finite period of time for the bus buffers to turn around. If a card drives data onto the expansion slot data bus immediately after PH0 rises, there may be a bus fight between the expansion card trying to drive the bus, and the Apple IIGS (or Apple IIe) bus buffers, which may not have turned around yet. A similar problem can occur if an expansion card leaves its read data on the bus too long after PH0 falls.

On the Apple IIGS, the data buffers turn around in 30 nanoseconds or less from the PH0 edges. Developers can avoid bus fights by simply using 74LS or 74HCT series parts and relying upon typical delay stackups to delay driving the data bus for approximately 30 nanoseconds. A more

solid technique is using the first rising edge of the 7M clock, after PH0 rises. This method may require an additional flip-flop, but it guarantees the desired delay. On the other hand, expansion card read data buffers should be turned off as soon as possible when PH0 falls to avoid a fight when the data buffers turn back out again. Figure 3 shows the recommended data transfer timing for the data bus.

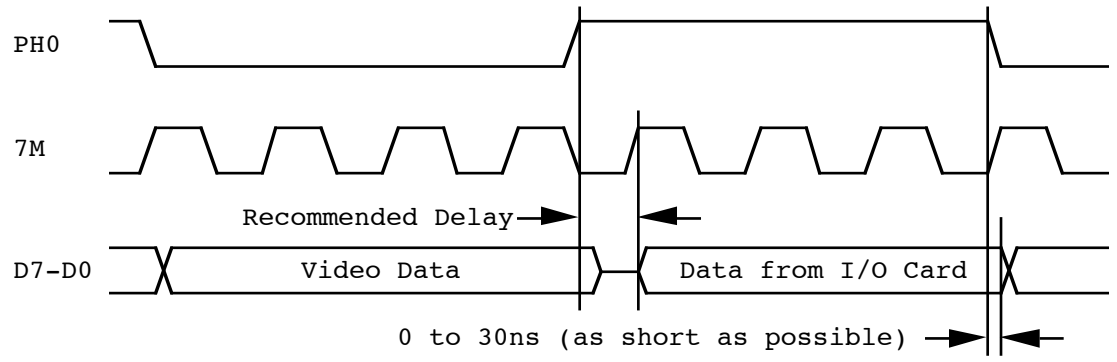

Figure 3–Recommended Data Transfer Timing

Ground Noise

Since the Apple II expansion slots were designed with only one ground pin, complex expansion cards sometimes have problems with excessive ground noise—especially in the IIGS, where the signals typically have faster rise and fall times. To reduce ground noise as much as possible, it is helpful to bypass all four supply voltages (+5 volt, +12 volt, –5 volt, –12 volt) to ground with electrolytic or solid tantalum capacitors, even if all the available voltages are not used on the expansion card. This additional bypassing has the effect of providing an improved ground by providing additional AC ground paths through the various supply pins.

To maintain a consistent ground quality over the board area on two-layer boards, it is important to properly grid the Vcc and ground traces and to fill in unused areas with ground plane.

Expansion Card Power Consumption

The Apple IIe and Apple IIGS expansion slot specifications indicate a total of 500 mA of +5 volt, 250 mA of +12 volt, 200 mA of -5 volt, and 200 mA of -12 volt power is available to all the expansion slots. With design improvements, the power required by disk drives has been reduced. Also, the Apple IIGS power supply is conservatively designed so there is somewhat more power available than indicated on the original specification. However, there is not unlimited power available, and expansion card developers should minimize power consumption as much as possible. Minimization can be accomplished by using CMOS wherever possible, using ROMs or RAMs with “power-down” mode when they are not enabled, and generally being careful to minimize parts count.

Since the Apple IIGS was released, several "super" expansion cards have become available. These cards typically provide a lot of performance and functionality, but in most cases, the power consumed by one card is more than the specified power available to **all** the expansion slots. Generally these cards work without problems. However, when several "super" cards are installed in a IIGS system, the total power drawn can exceed the available power supply capacity. This increase in power dissipation within the IIGS case can cause excessive heating and other associated problems when the internal case temperatures exceed the design specifications. This could conceivably damage the IIGS power supply. Please minimize the power requirements of expansion card designs wherever possible to avoid these problems.

Further Reference

- *Apple IIGS Hardware Reference*
- *Apple IIGS Firmware Reference*
- Apple IIGS Technical Note #28, Interface Card Design Guidelines
- Apple IIGS Technical Note #32, /INH Line Anomaly

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#69: The Ins and Outs of Slot Arbitration

Revised by: Matt Deatherage May 1990
Written by: Matt Deatherage September 1989

This Technical Note discusses the concept of a 14-slot Apple IIGS system through dynamic software slot arbitration. It presents concepts of which all IIGS programmers should be aware for full compatibility.
Changes since September 1989: Removed the section which stated that this Note showed how to switch slots in a way that does not interfere with slot arbitration and replaced it with the proper description, which is how to search a 14-slot system for peripherals and their identification bytes.

History

The Apple II has always had seven slots. In some cases (e.g., IIe), one of the slots was handled specially by the hardware, or (e.g., IIc) there was no hardware present for peripheral cards at all. But there have always been seven "slots" with firmware at location $Cn00 (where n is the slot number). If there was no firmware, there was no peripheral connected.

With the introduction of the Apple IIGS, the Apple II family saw its first 14-slot system. Seven hardware slots are provided for peripheral cards (like on the IIe), and seven internal "ports" with connectors on the back panel are provided by the system (like on the IIc). Since $C800 and above cannot be used for additional slots (that space is shared between all interface cards), each of the seven internal ports is matched with one of the slots, and either the port or the slot is enabled at any given time. The IIGS hardware allows switching between the two, so all fourteen slots could be used more or less simultaneously.

This situation posed a problem—the Apple II had only a disk operating system, not an overall operating system. Access to non-disk devices (i.e., character devices, like a serial card) was not arbitrated by the system in any way. The world was used to seven, and only seven, slots. Attempting to use more in a shared system such as the IIGS resulted in somebody jumping to slot firmware that somebody else had switched out. This tended to crash the system.

Then came GS/OS. With its centralized mechanism for dispatching to **all** devices connected to a system, GS/OS provides hope (for the first time) that a central routing mechanism can dynamically arbitrate between slots and ports, allowing the use of all 14 at one time. This is called dynamic slot arbitration, and is handled by a portion of GS/OS referred to as the Slot Arbiter.

Although the Slot Arbiter does not function in System Software 5.0 or earlier, it may function in the future. A skeleton is present in version 5.0 and later that accepts Slot Arbiter calls, but the skeleton does not actually switch any slots. This Note details the Slot Arbiter functionality and shows how to search a 14-slot system for peripherals and their identification bytes.

Note: The Slot Arbiter must **not** be used unless GS/OS is the current operating system.

The Slot Arbiter

The Slot Arbiter is accessed through the GS/OS system service call vector `DYN_SLOT_ARBITER` ($01FCBC). On ROM 03 and later, the vector is duplicated at $E10208. Entry to the Slot Arbiter is via a `JSL` instruction, and exit is via `RTL`. The parameters are as follows:

```
Entry:
    A = Slot to be selected (defined below)
    X = Undefined (or Bit Encoded Slot Configuration)
    Y = Undefined
    B = Undefined
    D = Undefined
    P = N V M X D I Z C  E
        x x 0 0 0 x x x  0

Exit:
    A = Error Code
    X = Bit Encoded Slot Configuration
    Y = Undefined
    B = Unchanged
    D = Undefined
    P = N V M X D I Z C  E
        x x 0 0 0 x x 0  0        If A = $0000 (no error)
        x x 0 0 0 x x 1  0        If A = $0010 (slot not available)
```

The slot number in the `A` register tells the Slot Arbiter what you are requesting. Bits 0-2 are the slot number in the range 0 through 7. Bit 3 is set if you are requesting an external slot and clear if you are requesting an internal port. Taken together, bits 0-3 give slot numbers of $0-$7 for internal ports and $9-$F for external slots. This is the same way that slot numbers are returned by the GS/OS `DInfo` command.

Bits 8 and 9 of the slot number indicate the action you wish the Slot Arbiter to take. A value in these two bits of 00 asks the Slot Arbiter to switch in the slot identified in bits 0 through 3. If both bits are set to 11, the Slot Arbiter restores all the slots to match the Bit Encoded Slot Configuration present in the `X` register. Bit Encoded Slot Configurations are discussed in the next section of this Note. Values other than 00 or 11 in bits 8 and 9 are **reserved** and must **not** be used by applications.

Bit 15 of the slot number is set if the slot selection has no slot dependencies. When the Slot Arbiter is asked to switch in a slot with no slot dependencies, it does no actual switching, although it returns a Bit Encoded Slot Configuration in the `X` register. The slot number and the definitions of the individual bits are illustrated in Figure 1.

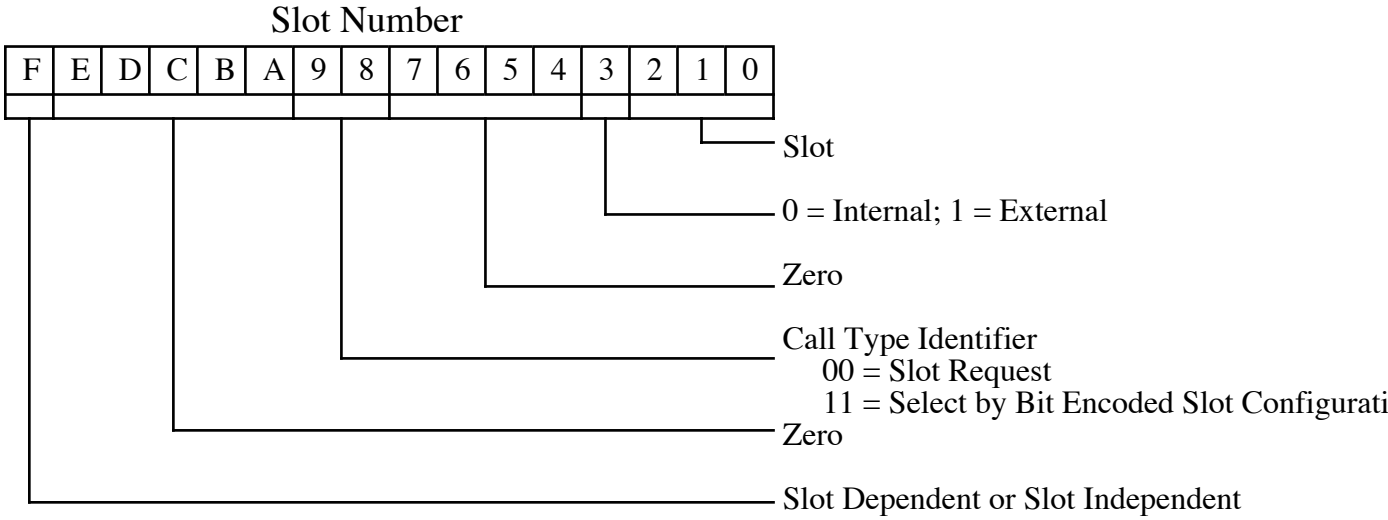

Figure 1–Slot Number and Bit Definitions

Bit Encoded Slot Configurations

Every call to the Slot Arbiter returns (on exit) a miniature picture of the slot configuration in the X register (as it was on entry). This picture has one bit set for each of the 14 slots; if the bit is set, then the corresponding slot is switched in. Bits 0 and 8 are reserved and are always clear. This picture is called a Bit Encoded Slot Configuration.

Since each external slot has the same number as an internal port (with bit 3 set), and since such pairs share the same address space, it follows that both of them may not be enabled at the same time. For example, port 5 and slot 5 ($D) both may not be enabled. This makes the high byte of the Bit Encoded Slot Configuration the eXclusive-OR of the low byte (excluding bits 0 and 8, which are always clear). Figure 2 illustrates the Bit Encoded Slot Configuration.

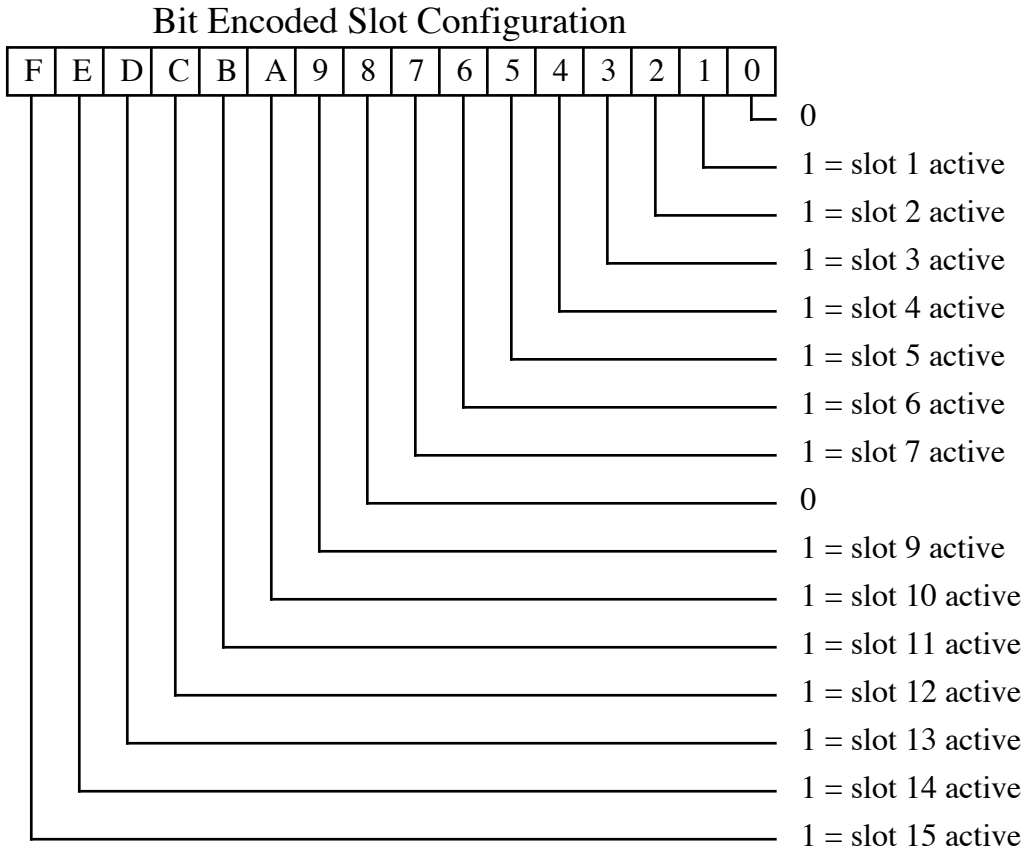

Figure 2–Bit Encoded Slot Configuration

By fully using the slot number parameter, the Slot Arbiter returns any aspect of the current slot configuration. Following are a few examples:

Slot number	Action Taken by Slot Arbiter
$8000	Returns current Bit Encoded Slot Configuration in the X register. This number asks the Slot Arbiter to switch in with no slot dependencies (no switching), so it just returns the Bit Encoded Slot Configuration.
$0300	Restore from Bit Encoded Slot Configuration. This command, when paired with the one above, can be used to save and restore a slot environment.
$0005	Asks the Slot Arbiter for internal port 5.

The Impact on Applications and Drivers

Applications which correctly do all input and output through GS/OS are affected by slot arbitration, except that they find more devices available. GS/OS uses the slot number parameter in the Device Information Block to call the Slot Arbiter, making sure the slot is available for the

device before it gets control. However, there are some applications (such as peripheral card configuration programs) which go directly to firmware or hardware, not using GS/OS. Perhaps the card has no ROM, so there is no generated driver, or perhaps there is no loaded driver and the generated driver does not control certain aspects of the hardware. In any case, such applications are directly impacted by slot arbitration.

Slot Searching

The first problem is finding the hardware. In a 14-slot system, it's not suitable to just look for ID bytes between $C100 and $C700—two peripherals may be sharing each of those pages of slot ROM space. Drivers must examine all 14 slots, with the aid of the Slot Arbiter. The following sample code demonstrates this technique:

```
find_slot          lda     #$8000                  ; request current Bit Encoded Slot
Configuration
                   jsl     slot_arbiter
                   phx                         ; save it on the stack

                   lda     #$000F              ; start with slot 15
                   sta     slot_number         ; be sure of the data bank when doing this!

slot_search        lda     slot_number         ; get the slot number to examine
                   jsl     slot_arbiter        ; and ask for it
                   bcs     continue_search     ; if an error, then don't look here
                   jsr     check_for_hw        ; this routine looks for your hardware
                   bcc     found_my_hw         ; if found it, we're done searching
continue_search    dec     slot_number         ; try the next lower slot
                   bpl     slot_search         ; (if there are any left, of course)

found_my_hw         plx                           ; get Bit Encoded Slot Configuration from
stack
                     lda    #$0300                ; and tell the Slot Arbiter to restore from
it
                   jsl     slot_arbiter

; We're done.  Our slot number is in the location slot_number.
```

Note: You **must** restore the previous slot configuration when searching for a slot. This is vital to device drivers during the `Drvr_Startup` call, and failure to do so at other times may break older, seven-slot applications.

The Slot Arbiter attempts to maintain a static seven-slot system for applications as reflected by the user's Control Panel settings. This system allows older applications to continue to work, as something they find in an older, seven-slot scan is still present. Newer applications may wish to consider implementing a 14-slot scan, but any slot not present in the static seven-slot environment requires a call to the Slot Arbiter before and after **every** access to that device. The overhead in such instances may be intolerable. Apple recommends that if an application requires hardware that cannot be found in a seven-slot scan, it request the user to set the Control Panel to make the hardware available and restart the system.

Using Slot-Dependent Hardware

Applications which have slot dependencies must call the Slot Arbiter before each use of the slot in question. Since Slot Arbitration changes the environment to which Apple IIGS programs have

become accustomed, everyone has a better chance of working by sticking to the general Apple IIGS rule of "put back what you use when you're done with it." Ask for the slot, use it, then restore the previous Bit Encoded Slot Configuration. (If you use multiple slots, you might wish to get the Bit Encoded Slot Configuration, save a copy, modify it to reflect the slots you want, and restore from the modified version.)

Note: Peripherals accessed through GS/OS do not have to call the Slot Arbiter; GS/OS handles this task automatically.

There are certain applications with more specialized needs, such as high-speed, single character input or output. In such cases, the Slot Arbiter may be a bottleneck. When a slot is not switched, the Slot Arbiter returns quickly, but when a slot must be switched, it takes a significant amount of time. Doubling that significant time for switching in and restoring gives a substantial overhead for each hardware access, which may be too much for some applications.

Note: It is **far** better to write a GS/OS driver to deal with hardware than to write a slot-dependent application to control it. A slot-dependent application must deal with the Slot Arbiter, and the user must quit the current application to run your application just to change some aspect of the hardware. Writing a GS/OS driver lets any application, desk accessory, or CDev control your hardware with regular GS/OS calls.

Problems with Slot-Dependent Tools

Code designed before the Slot Arbiter may have slot-dependencies that cause unexpected problems when dynamic slot arbitration is fully implemented. This list includes some of the Apple IIGS System Software. Specifically, the Text Tools and the `FWEntry` call in the Miscellaneous Tools present problems with dynamic slot arbitration.

Text Tools

When using the Text Tools to specify a device for input, output, or error, the value specified (a four-byte parameter) is assumed to be a slot number if it is in the range 0-7. The Text Tools were not designed to use Slot Arbiter-style slot numbers, and this causes a compatibility problem.

The Text Tools were modified in System Software 5.0 to recognize Slot Arbiter-style slot numbers where possible. The trick is that it's not possible as often as we'd like. External slots are specified by using slot numbers 9 through 15; if such a slot number is used as input to a Text Tools call, the appropriate Slot Arbiter call is made and that external slot is used if it can be made available. However, internal port numbers are in the range 1-7—the same range used by the old Text Tools to indicate which of two peripherals was switched in for a particular slot. The Text Tools cannot assume that you are requesting an internal slot when using a slot number between one and seven.

For example, your old application might do a seven-slot search and find a parallel printer card in slot 1 (where the Control Panel setting for that slot is "Your Card"). If the Text Tools assumed all slot numbers in the range one through seven meant internal ports, your application would actually access the internal port 1 firmware every time it tried to access the parallel card it found in slot 1; this problem occurs since old applications don't know and don't care about internal or external slots.

The Text Tools may be used to access any external slot (if available), but they may only be used to access internal ports that are set to internal in the Control Panel. The Text Tools slot numbers zero through seven always match the Control Panel settings.

Apple strongly recommends that the Text Tools not be used. GS/OS character-based drivers are preferable for standard character input and output. The Text Tools may be used for specialized purposes; however, you cannot access some internal ports and other components of the system that are not well-behaved. Doing so could cause your application to trash memory or media. You must assume these risks when using the Text Tools.

FWEntry

The Miscellaneous Tools call `FWEntry` should not be used to access entry points on a peripheral card (entry points in the $Cxxx range). As discussed, a poorly-behaved routine could switch the slot from one you've identified to something else between the time you identify the slot and issue the `FWEntry` call. Furthermore, the space between $C800 through $CFFF cannot be identified as belonging to any given slot, and the Slot Arbiter more or less guarantees that it won't be what you expect. Accesses to peripheral card ROM space ($Cxxx) should **only** be made by GS/OS drivers. `FWEntry` must **not** be used to access $Cxxx addresses.

`FWEntry` is still safe to use for addresses in the $D000-$FFFF range.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2
- *Apple IIGS Firmware Reference*
- *Apple IIGS Hardware Reference*
- *GS/OS Reference*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#70: Fast Graphics Hints

Written by: Don Marsh & Jim Luther September 1989

This Technical Note discusses techniques for fast animation on the Apple IIGS.

QuickDraw II gives programmers a very generalized way to draw something to the Super Hi-Res screen or to other parts of Apple IIGS memory. Unfortunately, the overhead in QuickDraw II makes it an unacceptable tool for all but simple animations. If you bypass QuickDraw II, your application has to write pixel data directly to the Super Hi-Res graphics display buffer. It also has to control the `New-Video` register at $C029, and set up the scan-line control bytes and color palettes in the graphics display buffer. Chapter 4 of the *Apple IIGS Hardware Reference* documents where you can find the graphics display buffer in memory and how the scan-line control bytes, color palettes, and pixel data bytes are used in Super Hi-Res graphics mode. The techniques described in this Note should be used with discretion—we do **not** recommend bypassing the Apple IIGS Toolbox unless it is absolutely necessary.

Map the Stack Onto Video Memory

To achieve the fastest screen updates possible, you must remove all unnecessary overhead from the instructions that perform graphics memory writes. The obvious method for achieving sequential writes to the graphics memory uses an index register, which must be incremented or decremented between writes. These operations can be avoided by using the stack. Each time a byte or word is pushed onto the stack, the stack pointer is automatically decremented by the appropriate amount. This is faster than doing an indexed store followed by a decrement instruction.

But how is the stack mapped onto the graphics memory? The stack can be located in bank $01 instead of bank $00 by writing to the `WrCardRAM` auxiliary-memory select switch at $C005. Bank $01 is shadowed into $E1 by clearing bit 3 of the `Shadow` register at $C035. Under these conditions, if the stack pointer is set to $3000, the next byte pushed onto the stack is written to $013000, then shadowed into $E13000. The stack pointer is automatically decremented so the stage is set for another byte to be written at $E12FFF.

Warning: While the stack is mapped into bank $01, you may **not** call **any** firmware, toolbox or operating system routines (ProDOS 8 or GS/OS). Don't even think about it.

Unroll All Loops

Another source of overhead is branching instructions in loops. By “straight-lining” the code to move up a scan-line’s worth of memory at one time, branch instructions are avoided. Following is an example of this technique.

```
lda    |164,y              ; accumulator is 16 bits for
pha                        ; best efficiency
lda    |162,y
pha
lda    |160,y
pha
```

In this example, the `Y` register is used to point to data to be moved to the graphics memory, and hard-coded offsets from the `Y` register are used to avoid register operations between writes.

Hard-Code Instructions and Data

In desperate circumstances, it is necessary to remove overhead from the previous code example. This can be accomplished by hard-coding pixel data into your code instead of loading pixel values from a separate data space and transferring them to the graphics memory (as in the example). If you are writing an arbitrary pattern of three or fewer constant values to the screen, for example, the following method is the fastest known:

```
lda    #val1
ldx    #val2
ldy    #val3
pha                        ; arbitrary pattern of pushes
phx
phy
phy
phx
```

In cases where many different values must be written to the screen, pixel data can be written to the screen using immediate push instructions:

```
pea    $5389               ; some arbitrary pixel values
pea    $2378
pea    $A3C1
pea    $39AF
```

Your program can generate this mixture of `PEA` instructions and pixel data itself, or it could load pixel data that already has `PEA` instructions intermixed (thus increasing the data size by one half).

Be Aware of Slow-Side and Fast-Side Synchronization

Estimating execution speed by counting instruction cycles is always a challenging task on the IIGS, but it is particularly tricky when one is writing to the graphics memory. The graphics memory resides in the side of the IIGS system controlled by the 1 MHz Mega II chip, which means that during all writes to this memory, the fast side of the system controlled by the Fast Processor Interface (FPI) chip must be synchronized with slow side of the system controlled by the Mega II, even if the system is running code at full native speed. This synchronization is performed automatically and transparently by the FPI in the IIGS, and it isn't normally of concern to the programmer. Animation programmers must worry about synchronization delays, however, because slight changes in graphics update code may change the frequency of these delays, and hence the speed of the program. In practical terms, this means that one loop writing

data to the graphics memory may run at the same speed as a second loop with a higher cycle count.

A careful analysis of the synchronization problem leads to the following tables, which are useful as a rough estimate of the speed attained by different pieces of code. Each entry is based on the number of cycles consumed during consecutive write instructions. For example, a series of `PEA` instructions requires five cycles for each 16-bit write. A short `PHA` instruction followed by a branch requires six cycles for each 8-bit write.

Fast Cycles per Write (byte)	Actual Speed (*µ*sec./byte)
3 to 5	2.0
6 to 8	3.0
9 to 11	4.0

Fast Cycles per Write (word)	Actual Speed (*µ*sec./word)
4 to 6	3.0
7 to 8	4.0
9 to 11	5.0

The times given in the tables apply only if the same number of fast cycles separate each consecutive write operation. The first write operation in a set of write instructions usually takes longer than subsequent writes, because the potentially long synchronization operation is accomplished at that time. Unpredictable delays caused by memory refresh slow things down further, although refresh delays byte-wide writes more often than word-wide writes. Therefore, it is usually preferable from a speed standpoint to use word-wide writes to the graphics memory.

For more information on synchronization cycle timing within the IIGS, see Chapter 2 of the *Apple IIGS Hardware Reference* and Apple IIGS Technical Note #68, Tips for I/O Expansion Slot Card Design.

Use Change Lists

The timing data given in the preceding section shows that it is not possible to perform full-screen updates in the time it takes the IIGS to scan the entire screen. In fact, it would be difficult to update more than one-sixth of the screen in one scan time. Therefore, it is necessary to update only those pixels which have actually changed from the previous frame of animation. One method of doing this is to precalculate the pixels which change by comparing each frame against the preceding frame. For interactive animation, fast methods must be developed for predicting which areas of the screen must be updated (a determination of the exact pixels might require more computation than the actual update would require).

Using the Video Counters

To achieve “tear-free” screen updates, it is necessary to monitor the location of the scan-line beam when writing to graphics memory. As described in Apple IIGS Technical Note #39, Mega II Video Counters, the `VertCnt` and `HorizCnt` Mega II video counter registers at $C02E-C02F allow you to determine which scan line is currently being drawn.

By using only the `VertCnt` register and ignoring the low bit of the 9-bit vertical counter stored in `HorizCnt`, you can determine within 2 scan lines which scan line is currently being drawn. The `VertCnt` video counter contains the number of the current scan line divided by two, offset by $80. For example, if the scan-line beam was currently refreshing either scan line four or five, `VertCnt` would contain $82 (4/2 + $80 or 5/2 + $80). Vertical blanking happens during `VertCnt` values $7D through $7F and $E4 through $FF.

Clever updates can modify twice as many pixels on the screen by sacrificing some smoothness, running at 30 frames per second instead of 60. The technique is as follows:

1. Wait for the scan line beam to reach the first scan line.
2. Start updates from the top of the screen, being careful not to pass the scan line beam.
3. Continue updates while the scan line beam progresses toward the bottom of the screen, then goes into vertical blanking, then restarts at the top of the screen.
4. Finish the update before the scan line beam catches the update point.

Careful use of this method allows a frame to be updated during two scans of the screen instead of just one. If you are not sufficiently careful, tearing results.

Note: The Apple IIGS main logic board Mega II-VGC registers and interrupts are not synchronous to the Apple II Video Overlay Card video and therefore should not be used for time synchronization with the Apple II Video Overlay Card video output. However, they can be used for time synchronization with the Apple IIGS video output. See the *Apple II Video Overlay Card Development Kit* for more information.

Interrupts

It is not possible to support interrupts while sustaining a high graphics update rate, unless jerkiness or tearing is acceptable. Be aware that many system activities such as GS/OS and AppleTalk depend on interrupts and do not function if interrupts are disabled.

Further Reference

- *Apple IIGS Firmware Reference*
- *Apple IIGS Hardware Reference*
- *Apple II Video Overlay Card Development Kit*
- Apple IIGS Technical Note #39, Mega II Video Counters
- Apple IIGS Technical Note #40, VBL Signal
- Apple IIGS Technical Note #68, Tips for I/O Expansion Slot Card Design

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#71: DA Tips and Techniques

Revised by: Dave "Mr. Tangent" Lyons May 1992
Written by: Dave Lyons November 1989

This Technical Note presents tips and techniques for writing Desk Accessories.
Changes since December 1991: Reworked discussion of NDAs and Command- keystrokes. Marked obsolete steps in "NDAs Can Have Resource Forks."

Classic Desk Accessory Tips and Techniques

Reading the Keyboard

For a CDA that runs only under GS/OS, the Console Driver is the best choice for reading from the keyboard. Other CDAs have two cases to deal with: the Event Manager may or may not be started. The Text Tools can read the keyboard in either case, but you should avoid using the Text Tools whenever possible (see Apple IIGS Technical Note #69, The Ins and Outs of Slot Arbitration).

You can call `EMStatus` to determine whether the Event Manager is started. When it is, you can read keypresses by calling `GetNextEvent`. When the Event Manager is not started, you can read keys directly from the keyboard hardware by waiting for bit 7 of location $E0C000 to turn on. When it does, the lower seven bits represent the key pressed. Once you've detected a keypress, you need to write to location $E0C010 to remove the keypress from the buffer.

Alternately, you can use `IntSource` (in the Miscellaneous Tools) to temporarily disable keyboard interrupts and then read the keyboard hardware directly. Be sure to reactivate keyboard interrupts if, and only if, they were previously enabled.

Just One Page of Stack Space

CDAs normally have only a single page of stack space available to them (256 bytes at $00/01xx). Your CDA may or may **not** be able to allocate additional stack space from bank 0 during execution. The following code (written for the MPW IIGS cross-assembler) shows a safe way to try to allocate more stack space and to switch between stacks when the space is available.

If ProDOS 8 is active, your CDA **cannot** allocate additional space (and there is no completely safe way to "borrow" bank 0 space from the ProDOS 8 application).

```
HowMuchStack   gequ    $1000          ;try for 4K of stack space

start          phd
               phb
               phk
               plb
               pha                    ;Space for result
               pha
               PushLong #HowMuchStack
               pha
               _MMStartUp
               pla
               ora    #$0f00          ;OR in an arbitrary auxiliary ID
               pha
               PushWord #$C001        ;fixed, locked, use specified bank
               PushLong #0            ;(specify bank 0)
               _NewHandle
               tsc
               sta    theOldStack
               bcs    NoStackSpace    ;still set from _NewHandle
               tcd
               lda    [1]
               tcd
;               clc                     ;carry is already clear
               adc    #HowMuchStack-1
NoStackSpace   pha
               ldx    #$fe
keepStack      lda    >$000100,x
               sta    stackImage,x
               dex
               dex
               bpl    keepStack
               pla
               tcs
               jsl    RealCDAentry     ;carry is clear if large stack available
               php
               php
               pla
               sta    pRegister
               sei
               ldx    #$fe
restoreStack   lda    stackImage,x
               sta    >$000100,x
               dex
               dex
               bpl    restoreStack
               lda    theOldStack
               tcs
               lda    pRegister
               pha
               plp
               plp
               lda    1,s
               ora    3,s
               beq    noDispose
               _DisposeHandle
               bra    Exit
noDispose      pla
               pla
Exit           plb
               pld
               rtl
pRegister     ds 2
theOldStack   ds 2
stackImage    ds.b 256
```

When this routine calls `RealCDAentry`, the carry flag is set if no extra stack space is available. If the carry is clear, the additional stack space was available and the direct-page register points to the bottom of that space.

```
RealCDAentry    bcs     smallStack              ;if c set, only 1 page of stack is
available
                ...                         ; put something interesting here
                rtl

smallStack      _SysBeep
                rtl
```

Note that interrupts are disabled while the page-one stack is being restored; they are reenabled (if they were originally enabled) only **after** the stack pointer is safely back in page one.

Interrupts, Event Manager, Memory, and CDAs

Whether the Event Manager is active or not, the user hits Apple-Ctrl-Esc and usually gets to the CDA menu. It looks the same, but what happens internally is different affects what happens when your CDA allocates memory.

When the Event Manager is active (as it normally is while the user is running a Desktop application), hitting Apple-Ctrl-Esc posts a `deskAcc` event to the event queue. The CDA menu appears only when the application calls `GetNextEvent` or `EventAvail` with the `deskAcc` bit enabled in the event mask.

So with the Event Manager active, the CDA menu and individual CDAs are running in the "foreground"—no processor interrupt is being serviced, and the foreground application is stuck inside the `GetNextEvent` or `EventAvail` call. The Memory Manager knows that no interrupt is in progress, so it will happily compact and purge memory if necessary to carry out a memory allocation request from your CDA. This is just fine, since the foreground application made a toolbox call—unlocked memory blocks are not guaranteed to stay put.

When the Event Manager is not active, hitting Apple-Ctrl-Esc either enters the CDA menu immediately (if the system Busy Flag is zero) or calls `SchAddTask` so that the CDA menu appears during a the next `DECBUSYFLG` call that brings the system Busy Flag down to zero. If the CDA menu appears during a `DECBUSYFLG` , normal memory compaction and purging are possible, just like when the Event Manager is active.

But if the Busy Flag was zero when the user hit Apple-Ctrl-Esc, then the CDA menu appears inside of the interrupt, and the foreground application is at an unknown point where it may justifiably expect that unlocked memory blocks will not move or be purged (see *Apple IIGS Toolbox Reference*, Volume 1, page 12-5). (Note that the Desk Manager does a tricky dance to allow additional interrupts to occur, even though the Apple-Ctrl-Esc interrupt will not return until the user chooses Quit from the CDA menu. Normally interrupts cannot be nested; the Desk Manager and AppleTalk are exceptions.)

The Memory Manager knows an interrupt is in progress, so `CompactMem` takes no action and memory allocation requests do not cause unlocked memory blocks to move and do not attempt to purge purgeable blocks to make room. Memory allocation requests will still normally succeed, but you will not be able to allocate a block larger than the value returned by `MaxBlock`.

New Desk Accessory Tips and Techniques

An NDA Can Find its Menu Item ID

After the application has called `FixAppleMenu`, an NDA can look at its menu item template (after the "`\H`" in the NDA header) to determine the menu ID corresponding to the NDA's name in the Apple menu. This is sometimes useful to pass to `OpenNDA` (if the NDA has some way to open itself), or to pass to a Menu Manager call.

Finding the menu item ID in the NDA's header is easy if the NDA is written in assembly. In a high-level language it may be harder (if you don't have direct access to your NDA's header, you need to find it on the fly and scan for the "`\H`").

NDAs and Command- Keystrokes

To give the user a consistent way to close NDA windows, System 6.0 handles Command-W automatically when a system window is in front. It calls `CloseNDAbyWinPtr` without letting the NDA or the application see the Command-W.

However, there is a special action code (`optionalCloseAction`) that an NDA can accept to handle the Close request itself. This way the NDA can offer the user a chance to cancel the Close, which is impossible when the system calls the NDA's main Close routine, as `CloseNDAByWinPtr` does. (See the System 6.0 Toolbox documentation for details.)

There is no way for an NDA to accept **some** keystrokes and pass others along to applications, but if your NDA does not want any keystroke events, turn off the corresponding `eventMask` bits in the NDA header (this allows the application to receive keystrokes while your NDA window is in front).

Calling InstallNDA From Within an NDA

It is possible to write an NDA that installs other NDAs. However, with System Software 5.0 and later, `InstallNDA` returns an error when called from an NDA. When your NDA has control because the Desk Manager called one of your NDA's entry points, the Desk Manager's data structures are already in use, so `InstallNDA` is unable to modify them.

The solution is to use `SchAddTask` in the Scheduler to postpone the `InstallNDA` call until the system is not busy. Remember that the Bank and Direct Page registers are not defined when your scheduled task is executed.

Processing mouseUp Events

When an NDA's action routine receives a `mouseUp` event, it is not always safe for the NDA to draw in its window.

For example, when the user drags an NDA window, the NDA receives the `mouseUp` before the window is actually moved, and before `DragWindow` erases the outline of the new window

position, which may overlap the window's content. In addition, when the user chooses a menu item, the front NDA receives the `mouseUp` before the menu's image is removed, and the image may overlap the NDA's window. In either case, drawing in the window makes a mess.

The solution is to avoid drawing in direct response to a `mouseUp`. Instead, invalidate part of the window to force an update event to happen later.

NDAs Can Have Resource Forks

Following is the recommended way for a New Desk Accessory to use its file's resource fork.

In the NDA's Open routine, do the following. Steps that are obsolete (and safely omitted) with System Software 6.0 and later are marked with an asterisk (*):

1. Call `GetCurResourceApp` and keep the result.
2. If the NDA does not already know its Memory Manager user ID, call `MMStartUp` to get it.
3. Call `ResourceStartUp` using the NDA's user ID.
4. Call the Loader function `LGetPathname2` with the NDA's user ID (and a `fileNumber` of $0001) to get a pointer to the NDA's pathname. (The result is a pointer to a class-one GS/OS string.)
*5. Use `GetLevel` to get the current file level, then use `SetLevel` to set it to zero. This helps protect your resource fork from being closed accidentally.
6. Use `GetSysPrefs` to get the current OS preferences, then use `SetSysPrefs` to ensure that the user is prompted, if necessary, to insert the disk containing your resource fork. (To compute the new preferences word, take the current one, `AND` it with $1FFF, and `ORA` it with $8000. This tells GS/OS to deal with volume-not-found conditions by putting up a please-insert-disk dialog with an OK button and a Cancel button.)
7. Call `OpenResourceFile` using the result from `LGetPathname2`. Save the returned `fileID`—you need it when closing the file. (Be prepared to deal with an error, such as $0045, Volume Not Found.)
8. Use `SetSysPrefs` to restore the OS preferences saved in step six.
*9. Use `SetLevel` to restore the file level to its old value (saved in step five).
10. Call `SetCurResourceApp` with the old value saved in step one.

In the NDA's action routine, no special calls are necessary—the Desk Manager calls `SetCurResourceApp` automatically before calling your action routine, so your NDA's own resource search path is already in effect.

Run queue routines and NDA installs with `AddToRunQ` are treated the same way—the NDA's resource search path is automatically in effect when the run queue routine is called.

In the NDA's Close routine, do the following:

1. Call `CloseResourceFile` with the `fileID` that was returned when you opened it.
2. Call `ResourceShutDown` with no parameters.

NDAs Must Be Careful Handling Modal Windows

If your NDA uses its resource fork and calls `TaskMaster` with a restricted `wmTaskMask` to produce a modal window, you must be careful **not** to allow `TaskMaster` to update the contents of any application windows that happen to need updating.

The problem is that an application window's `wContDraw` routine can reasonably assume that the current Resource Manager search path is the application's, but `TaskMaster` does not take any special steps to set it. When the content-draw routine draws controls which were created from resources which are not presently in the resource search path, the system may crash.

If your NDA does not start up the Resource Manager, the Desk Manager is unable to `SetCurResourceApp` to your NDA, so the application's search path is still in effect—no problem. But if your NDA **does** start the Resource Manager, you have to be careful not to cause application routines to be called.

Avoid Hard-Coding Your Pathname

If your NDA needs to know its own pathname or the pathname of the directory it's in, call `LGetPathname` or `LGetPathname2` using your User ID. This is a better method than hard-coding "*:System:Desk.Accs:MyDAName" because the user may change your DA's file name or use a utility to install it from some non-standard directory.

Avoid Extra GetNewID calls

Normally there is no reason for a Desk Accessory to call `GetNewID`. When you can, just call `MMStartUp` to find your own User ID, and use that. You can freely use all the auxiliary IDs derived from your main ID (`MMStartUp`+$0100, `MMStartUp`+$0200, ..., `MMStartUp`+$0F00).

By not calling `GetNewID`, you conserve the limited supply of IDs (255 of in the $50xx range for Desk Accessories), and you make life easier for people trying to debug their systems, since all your allocated memory can be readily identified.

Open is Not Called if NDA is Already Open

Your NDA's Open routine does not get called if the user chooses the NDA from the Apple menu while the NDA is already open. In this case, the Desk Manager simply calls `SelectWindow` on your existing window.

There is no need to include code in your Open routine to check if your window is already open, and to call `SelectWindow` if it is.

Further Reference

- *Apple IIGS Toolbox Reference,* Volumes 1-3
- *GS/OS Reference*

- *Apple IIGS Hardware Reference*
- Apple IIGS Technical Note #53, Desk Accessories and Tools
- Apple IIGS Technical Note #57, The Memory Manager and Interrupts
- Apple IIGS Technical Note #69, The Ins and Outs of Slot Arbitration

Apple IIGS
#72: QuickDraw II Quirks

Revised by: Dave Lyons May 1992
Written by: Dave Lyons & C. K. Haun <TR> November 1989

This Technical Note points out some things things you need to watch out for when using QuickDraw II, especially with FastPort-aware and Shadowing modes.

Changes since November 1990: Removed some obsolete information on `ScrollRect` and on shadowing. Noted that `DrawPicture` in 6.0 is now compatible with FastPort mode. Added a warning about making QuickDraw II calls while QuickDraw II is not started.

Don't Call QuickDraw II While It's Inactive

Most QuickDraw II functions behave unpredictably if you call them while QuickDraw II is inactive, so watch it! Don't make QuickDraw II calls while QuickDraw II isn't started, except as documented. `GrafOn` and `GrafOff` are okay. (And so are `QDStartUp`, `QDVersion`, and `QDStatus`.)

FastPort-Aware Anomaly

Before System 6.0, when the FastPort-aware bit is turned on in the `MasterSCB` parameter to `QDStartUp`, `DrawPicture` did not notice changes in the pen pattern. If your application does not require 6.0 and uses pictures, either directly or indirectly (i.e., by printing to the ImageWriter driver), you may need to leave FastPort-aware mode turned off to get the expected behavior.

FastFont and Large Pixel Maps

FastFont does not work correctly when drawing past the first 64K of a pixel map. If you are drawing text that uses FastFont (i.e., Shaston 8), you can avoid this problem by using a non-rectangular `clipRgn`.

Don't ShowPen While Collecting Polygons, Regions, or Pictures

The Macintosh QuickDraw documentation permits calling `ShowPen` after an `OpenPoly`, `OpenRgn`, or `OpenPicture` call to cause drawing calls to contribute to a polygon, region, or picture **and** draw to a pixel map at the same time.

The Apple IIGS QuickDraw II documentation does **not** say you can do that. In some cases, it works, but it works "by accident" and it's not one of the things Apple tests or guarantees in QuickDraw II.

You May Need SetBufDims!

The call description for `SetBufDims` on page 16-215 of Volume 2 of the*Toolbox Reference* is misleading. The note in the description states, "You only need to make this call if your application is going to use, or allow the user to choose, fonts that have unusually large values of `chExtra` and `spExtra`." This is **not** true; you need to call `SetBufDims` to adjust the clipping buffers for your application if you plan to use a `clipRgn` that has a greater width than the width you passed at `QDStartUp`.

`SetBufDims` sets the clipping buffer width as well as that of the text buffer, so if you plan to use a clipping region larger than the startup port width you must use `SetBufDims`.

Be aware that this call may be necessary even if your application does not ever set a clipping region or rectangle. Some toolbox calls assume that the clipping buffer size is correct based on the parameters passed to that routine. For example, if the `locInfo` you pass to `CopyPixels` has a `width` parameter that is wider than the width you passed at `QDStartUp`, `CopyPixels` may fail. A safe rule of thumb is to make sure (possibly by setting) that the `width` parameter in the buffer dimensions is the same or greater than the widest `width` in the `locInfo` structures passed to routines that use them.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1 and 3

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#73: Using User Tool Sets

Revised by: Dave "flag bits" Lyons July 1991
Written by: Dave Lyons November 1989

This Technical Note explains how to write a user tool set and why writing a user tool set is better than stealing a system tool set number.

Changes since January 1991: Expanded recommendation on where to keep user tool set files on disk and clarified `SetTSPtr` information.

The *Apple IIGS Toolbox Reference* describes system tool sets, which are usually called through the system tool dispatcher vectors 1 ($E10000) and 2 ($E10004).

There are 255 possible system tool set numbers (1 through 255). All of these are reserved for definition by the system. If your program is "borrowing" a system tool set number, please feel guilty and switch over to the user tool set numbers. There are 255 of them too, and they're called through user tool dispatcher vectors 1 ($E10008) and 2 ($E1000C). All 255 user tool set numbers are available for the current application to use as it chooses. (Desk accessories are forbidden to use user tool sets.)

Of the four tool dispatcher vectors, only the first one ($E10000) has received a lot of publicity. $E10008 works just like $E10000, except that it passes control to a user tool set instead of a system tool set.

The second vector of each pair ($E10004 and $E1000C) works just like the first, except that one extra `RTL` address must be pushed onto the stack after any parameters are pushed. This way you can have a subroutine to do some or all of your toolbox dispatching, and that subroutine can do extra processing before or after the tool call, or both.

How Can I Write a User Tool Set?

Appendix A of *Toolbox Reference*, Volume 2, shows how to write a user tool set. Your tool set's Work Area Pointer is a four-byte value you can set with `SetWAP` and get with `GetWAP`. The WAP value is already loaded into the `Y` and `A` registers every time one of your tool set's functions gets control. The traditional use for the WAP is to keep track of an area of memory owned by your tool set.

If you do use the WAP in a conventional way, your `xxxStatus` function should return `TRUE` if the WAP is nonzero; your `xxxStartUp` function should set the WAP to a non-zero value pointing to some memory space you own (provided by the caller, or allocated with `NewHandle` using a memory ID provided by the caller); and your `xxxShutDown` function should set the WAP back to zero.

Since the `X` register contains the tool set and function number when one of your functions gets control, it is not necessary for a tool set to be written to be used as a predetermined user tool set number. At execution time, your tool set can compute the proper error codes and values to send to `GetWAP` and `SetWAP`.

Note: At the bottom of page A-8 of the *Apple IIGS Toolbox Reference*, Volume 2, "`lda #$90`" should read "`lda #$8100`" for version 1.0 prototype. On page A-10, the figure should show **two** RTL addresses (6 bytes) on the stack.

ToStrip and ToBusyStrip Vectors

These two vectors are for tool sets to jump to when a function exits.

`ToBusyStrip`	$E10180
`ToStrip`	$E10184

Inputs: X = error code (0 if no error)
Y = number of bytes of input parameters to strip

When your function is ready to exit, set up the registers and jump to `ToStrip`. It shifts the six bytes of `RTL` addresses up by Y bytes, sets up A and the carry flag appropriately, and returns to whomever called the tool.

If the system busy flag needs to be decremented, jump to `ToBusyStrip` instead of `ToStrip`.

How Can I Load My Tool Set From Disk?

One way to load your tool set from disk is to use `InitialLoad` or `InitialLoad2`, supplying a pathname like "9:MyToolset" (prefix 9 is initially set to the directory containing your application; prefix 1 also works, but its length is limited to 64 characters). You can then use `SetTSPtr` to tell the Tool Locator about your tool set, as shown in Appendix A.

Note that `SetTSPtr` calls your `xxxBootInit` function. Even if there is no useful work to be done at BootInit time, you still need to have a `BootInit` function (function number 1) that returns $0000 in the Accumulator and the carry flag cleared..

When you're done with your tool set, call `UserShutdown` on the memory ID returned by `InitialLoad`, so the memory it's using is disposed of or made purgeable. (You can shut it

down and allow it to remain in memory in a purgeable state; if you do this, you should try to revive your tool set with `Restart` before you try `InitialLoad` or `InitialLoad2`.)

To allow several applications to share one copy of a user tool set file, you may want to keep your user tool set in the user's `*:System:Tools` folder. To avoid duplicate file names, leave the `ToolXXX` names for System tool sets, and give your user tool set a descriptive name.

If your tool set is not found in the `*:System:Tools` folder, you can then check the `9:` folder. This way users do not need to burden their `*:System:Tools` folders if few of their applications use a particular user tool set or if space on their boot volume is limited.

When your application quits and calls `TLShutDown`, the system disconnects your tool set from the user tool set TPT. If the `UserShutDown` is not followed immediately by the `TLShutDown`, you may wish to use `SetTSPtr` to cleanly remove your tool set from the system (set the tool set pointer so that it points at a zero word).

Note: Because of the way the tool dispatcher transfers control to toolbox functions, a function's entry point must not be at the first byte of a bank ($xx0000). This is normally not an issue, since it's common to put the actual code right after the function pointer table, all in one load segment. Just make sure no function begins at the first byte of a load segment, and you're safe.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 2
- *GS/OS Reference*

Apple IIGS
#74: Top Ten List Manager Things

Revised by: Dave Lyons May 1992
Written by: Jim Mensch November 1989

This Technical Note presents a method for speeding up custom List Draw routines, with sample source code for the APW assembler.

Changes since November 1989: Added information on `memFlag` and on shared `rListRef` resources, and noted that System 6.0 already checks the clip region and calls your `listDraw` routine only when needed.

Ten—More `memFlag` Bits

In each member record, bits 0 and 1 of `memFlag` indicate whether `memPtr` is a pointer, handle, or resource ID. You don't normally have to worry about that—a custom `listDraw` routine is one place that you do. The complete definition of `memFlag` is as follows:

Bit	Description
7	`memSelected`
6	`memDisabled`
5	`memNever` (Inactive)
4-2	reserved—set to zero
1-0	00 = `memPtr` is a pointer
	01 = `memPtr` is a handle
	10 = `memPtr` is a resource ID (type is `rPString` or `rCString`)
	11 = reserved

Nine—Sharing `rListRef` resources

When `listRef` is a resource ID, the List Manager calls `LoadResource` every time it needs your `rListRef` resource. If two or more lists share the same `rListRef`, they will get the same handle from `LoadResource` and will interfere with each other.

To give each list its own copy of your the `rListRef` resource, load the resource yourself and use `DetachResource`. Then feed the `listRef` to the List Manager as a handle. Repeat the process for each list.

Eight—Custom `listDraw` Routines and the Clip Region

The custom listDraw routine below speeds up your list when running System Software earlier than 6.0. The System 6.0 List Manager already calls your `listDraw` routine only for members that will not be completely clipped (but this is still a good starting point if you're writing a custom `listDraw` routine for some other reason).

To scroll text, the List Manager calls `ScrollRect` to scroll the list—then 6.0 redraws the newly-exposed members, and older versions redraw all the visible members. On small lists this is fine, but on larger lists it can cause the redrawing of much data that is already on the screen, which can take time. If your application does not require 6.0, you may want to use a custom `listDraw` routine like this one.

First, we check the current `clipRgn` (which the List Manager was kind enough to shrink down to include only the portion of the list that needs redrawing) against the passed item rectangle. If the rectangle is in any way enclosed in the `clipRgn`, then the member is redrawn; otherwise the routine simply returns to the List Manager without drawing. This sample routine is designed to work only with Pascal-style strings, but it can be easily modified to use any other type of string you choose.

```
MyListDraw   Start
;
; This routine draws a list member if any part of the member's
; rectangle is inside the current clipRgn.
;
; Note that the Data Bank register is not defined on entry
; to this routine.  If you use any absolute addressing, you
; must set B yourself and restore its value before exiting.
;
top          equ 0
left         equ top+2
bottom       equ left+2
right        equ bottom+2
rgnBounds    equ 2
;
oldDPage     equ 1
theRTL       equ oldDPage+2
listHand     equ theRTL+3
memPtr       equ listHand+4
theRect      equ memPtr+4
             using globals

             phd
             tsc
             tcd

             pha
             pha
             _GetClipHandle
             PullLong listHand

             ldy #2
             lda [listhand],y
             tax
             lda [listhand]
             sta listhand
             stx listhand+2

             lda [therect]               ; now test the top
             dec a                       ; adjust and give a little slack
             ldy #rgnbounds+bottom
             cmp [listhand],y            ; rgnRectBottom>=top?
```

```
            blt skip2
            brl NoDraw                  ; if not don't draw..
Skip2       ldy #bottom                 ; now see if the bottom is higher than the top
            inc a                       ; give a little slack
            lda [therect],y
            ldy #rgnBounds+top
            cmp [listhand],y
            blt NoDraw
NoTest      ANOP

            PushLong theRect
            _EraseRect              ; erase the old rectangle

            ldy #left
            lda [theRect],y
            tax
            ldy #bottom
            lda [theRect],y
            dec a
            phx
            pha
            _MoveTo
            ldy #2
            lda [memptr],y
            pha
            lda [memptr]
            pha
            _DrawString

            ldy #4
            lda [memPtr],y
            and #$00C0              ; strip to the 6 and 7 bits
            beq memDrawn            ; if they are both 0 the member is drawn
            cmp #$0080              ; member selected?
            bne noSelect            ; member not selectable
            PushLong theRect
            _InvertRect
            bra memDrawn
; if we get here the member is disabled
noSelect    PushLong #DimMask
            _SetPenMask
            PushLong theRect
            _EraseRect
            PushLong #NorMask
            _SetPenMask
memDrawn    ANOP


; exit here
            pld
            sep #$20
            longa off
            pla
            ply

            plx
            plx
            plx
            plx
            plx
            plx
            phy
            pha
            rep #$20
            longa on
            rtl
```

```
DimMask     dc  i1'$55,$AA,$55,$AA,$55,$AA,$55,$AA'
NorMask     dc  i1'$FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF'
            end
```

Seven through One—Reserved For Future Expansion

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1 and 3

Apple IIGS
#75: BeginUpdate Anomaly

Revised by: Dave Lyons May 1992
Written by: Eric Soldan January 1990

This Technical Note discusses a Window Manager anomaly with the handling of the `visRgn` and the `updateRgn` between `BeginUpdate` and `EndUpdate` calls.
Changes since January 1990: Updated for System 6.0. `CopyPixels` is in a static segment, and GS/OS automatically prompts for disks on the text screen when necessary to avoid interfering with a window update in progress.

If an application calls `BeginUpdate`, it needs to be fully aware of what is going on behind the scenes in terms of its `visRgn` and `updateRgn`. Typically an application has `TaskMaster` handle the update events. `TaskMaster` calls `BeginUpdate`, the application update procedure, then `EndUpdate`. So any application that uses `TaskMaster` to handle updates, whether or not it makes any `BeginUpdate` calls directly, needs to be aware of problem described in this Note.

`BeginUpdate` is responsible for intersecting the `visRgn` and the `updateRgn` and making the intersection of these two regions the temporary `visRgn`. (`EndUpdate` undoes this effect.) Following are the steps `BeginUpdate` takes to do this:

1. Localize the `updateRgn`. (All `grafPort` regions are local, therefore the `visRgn` is local. All window regions are global, therefore the `updateRgn` is global. One of them has to change if they are to be intersected correctly.)
2. Intersect the `visRgn` and localized `updateRgn`, then place the result in the `updateRgn`.
3. Swap the `visRgn` and `updateRgn` handles.

   The handle swapping has two effects:

   - Makes the intersection region the current `visRgn`.
   - Saves the real `visRgn` as the `updateRgn`. (Saving the real `visRgn` is necessary because everything has to be restored to normal by `EndUpdate`.)

`EndUpdate` restores things to normal after an update procedure is finished. When an application calls `EndUpdate`, it swaps back the handles and sets the `updateRgn` to empty.

So What's the Problem?

The problem is that the `updateRgn` is not a very good place to save the `visRgn`. Since `InvalRect` and `InvalRgn` modify the `updateRgn`, if either of these two calls is made between a `BeginUpdate` and `EndUpdate`, they modify the saved `visRgn`. When the update is finished, `EndUpdate` restores the modified `visRgn` instead of the original.

The solution to this problem seems simple enough: don't call `InvalRect` or `InvalRgn` between `BeginUpdate` and `EndUpdate`. Unfortunately, there are other calls which can call `BeginUpdate`, `EndUpdate`, `InvalRect`, and `InvalRgn`, so an application might inadvertently call one of these routines.

If this situation isn't bad enough already, you could really mess things up by opening another window between `BeginUpdate` and `EndUpdate` calls. Opening a window at this time may seem like a perfectly normal thing (i.e., to display an alert); however, opening a window forces the recalculation of the `visRgn` for any windows obscured by the new window. If the window being updated has its `visRgn` recalculated, the application obviously loses the `visRgn` that `BeginUpdate` created. This doesn't seem too serious since the `visRgn` is restored to the entire visible part of the window when the new window is closed; however, it does mean that the application would have to update the entire window instead of the original `updateRgn`.

Unfortunately, the Window Manager also posts update events for the portion of the window that was obscured, and it does this by changing the `updateRgn`. Of course the `updateRgn` for the window being updated is really the `visRgn` that is being "safely" preserved until the `EndUpdate` call. So, there are some really good reasons why this can't be done.

Okay, so along with not making calls to `InvalRect` and `InvalRgn` between `BeginUpdate` and `EndUpdate`, an application cannot open any other windows either. Good.

Now to make things even worse.

Starting with System 5.0, some toolbox functions are stored on disk in dynamic segments and loaded when they are first called. For example, `CopyPixels` is in a dynamic segment in System versions 5.0 through 5.0.3. If the startup disk is not available and the system prompts for it between `BeginUpdate` and `EndUpdate` by calling `AlertWindow`, the bad things discussed above happen.

Starting with System 6.0, the system is smart enough not to prompt for a disk using `AlertWindow` if a window update is in progress. (Internally, GS/OS calls `WindStatus` to see if it can prompt on the graphics screen. If `BeginUpdate` has been called more times than `EndUpdate`, `WindStatus` fibs by returning with the carry set. GS/OS takes the hint and prompts for the disk with a text dialog instead.)

But I Have to Do…

If you absolutely must do some of the things previously discussed, there is a way to accomplish it. It is not simple, but it can be done.

Assuming that `BeginUpdate` has been called, and an application is in its update procedure:

1. Create a new region and copy the `visRgn` into it. Doing this allows the application to restore the `visRgn` to just the area to be updated that `BeginUpdate` calculated. This needs to be done for any other windows which obscure a part the the window being updated. Again, these are not windows that an application would open directly. `CopyPixels` may open a window, since it is a dynamic segment and may need to get loaded from a disk that is off-line.
2. Create a new region, then swap its handle with the `updateRgn` handle. This protects the real `visRgn` and lets an application call `InvalRect` and `InvalRgn` at any time if necessary. It also means the application doesn't need to worry about the system making these calls either. The `updateRgn` is also an empty region after the swap, so any contributions to it constitute a valid update event that needs to be handled.
3. Do the update part of the update procedure. In this part, if the application has any calls to `CopyPixels`, or any other QuickDraw Auxiliary dynamic segment functions, after the call is completed, copy the saved `visRgn` back to the `visRgn` of the `grafPort`. The closing of the dynamic segment alert window recalculates the `visRgn`, and copying it undoes this effect. Do **not** do the same for the `updateRgn`. Leave the `updateRgn` alone. We are accumulating an actual `updateRgn`, and the closing of the alert window for the dynamic segment may have contributed to this region.

There are two methods for leaving the update procedure. Although the second method works whether or not an application uses `TaskMaster`, if an application does not use `TaskMaster`, then the first method is simpler.

The procedure without using `TaskMaster` (i.e., you made the `BeginUpdate` call, and you will make the `EndUpdate` call) is as follows:

A. Dispose of the region created in Step 1. This region was only needed to restore the partial `visRgn` that `BeginUpdate` calculated after a window was opened.
B. Swap the `updateRgn` handle with the region handle created in Step 2.
C. Make the `EndUpdate` call.
D. If the region created in Step 2 is not empty, copy this region into the `updateRgn` for the window with `CopyRgn`. You can't just do an `InvalRgn` with it because `InvalRgn` globalizes the region and the region is already global. You want to copy the region since this generates a valid update event. You can use `CopyRgn` instead of `UnionRgn` because the update region is empty.
E. Dispose of the region created in Step 2.

With `TaskMaster`, things are a little messier. Since `TaskMaster` makes the `EndUpdate` call, you have less control over the situation. It is important to do the `EndUpdate` before generating the update event. Posting the update event has to happen outside the update procedure, since you have to leave the update procedure for `TaskMaster` to do the `EndUpdate`. So it follows that you do Steps A and B, post an application event to handle the rest externally, and when the application event is handled, do Steps D and E.

Some consideration was given to posting an application event via the `PostEvent` call. Unfortunately, there is a possibility that this application event will drop out of the queue not handled. When the queue is full, the oldest event is dropped, and this could occur to application events, which would be very bad in this case. Due to this possibility, posting an application event refers to setting a global variable that is checked before the `TaskMaster` call in the main event loop. This can be considered equivalent to posting an event via the `PostEvent` call.

So, the `TaskMaster` case would be as follows:

A. Dispose of the region created in Step 1.
B. Swap the `updateRgn` handle with the region handle created in Step 2.
C. Store the handle of the region created in Step 2 in a global variable named `eventUpdateRgn`. Store the current window port in a global variable named `eventWindowPort`.
D. Return to `TaskMaster`, which returns to the main event loop.
E. Immediately after the `TaskMaster` call in the main event loop, check the global variable `eventUpdateRgn`. If it is not `NULL` then:
    a. Copy the region into the `updateRgn` of the window `eventWindowPort`. Using `CopyRgn` is the easiest way to copy the region. (Copying the region posts an update event if the event `UpdateRgn` is not `NULL`.
    b. Dispose of the region `eventUpdateRgn`, then set the variable `eventUpdateRgn` to `NULL`, so that this "event" won't be handled again.

Of course, the simplest way to handle all of this is to avoid situations where you have to take the steps described above. If things like opening a window (or allowing the system to open one) and `InvalRect` and `InvalRgn` can be avoided between calls to `BeginUpdate` and `EndUpdate`, so can all of this ugliness.

Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2

Apple IIGS
#76: Miscellaneous Resource Formats

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage, C.K. Haun, Llew Roberts & Dave Lyons January 1990

This Technical Note describes resource structure formats for previously-unpublished types.

Changes since December 1991: Added information on `rFont` resources. Clarified the note about `rVersion` resources to note that version numbers must increase with subsequent releases for the Finder's benefit.

The format used to describe the resources is similar to that used in File Type Notes, where the offsets, given in the form (+xxx), determine the offset from the beginning of the resource.

Sampled Sound Resource (Type: $8024, rSoundSample)

The following describes the Sampled Sound resource format. It consists of a ten-byte header followed by the sample data bytes.

Format	(+000)	**Word**	This must always be zero.
Wave Size	(+002)	**Word**	Sample size in pages (256 bytes per page). For example, an 8K sample takes 32 pages; a 128K sample requires $200 pages.
Rel Pitch	(+004)	**Word**	The high byte of this word is a semitone value; the low byte is a fractional semitone. These values are used to tune the sample to correct pitch. See HyperCard IIGS Technical Note #3, Pitching Sampled Sound.
Stereo	(+006)	**Word**	The output channel for this sound is in the low nibble of this word.
Sample rate	(+008)	**Word**	The sampling rate of the sound, in Hertz (Hz).
Sound	(+010)	**Bytes**	The sampled sound data. The bytes are all 8-bit samples. The sample starts here and continues until the end of the resource.

The resource compiler template follows:

```
#define rSoundSample  $8024

/*---------------------- rSoundSample --------------------*/
type rSoundSample {
       integer;              /* format */
       integer;              /* wave size */
       hex integer;          /* rel pitch */
       integer;              /* stereo channel */
       unsigned integer;     /* sample rate */
       hex string;           /* raw 8 bit sound data */
};
```

Cursor Resource (Type: $8027, rCursor)

The following describes the Cursor resource format:

height	(+000)	**Word**	The height of the cursor, in pixels.
width	(+002)	**Word**	The width of the cursor, in **Words**.
image	(+004)	**Bytes**	The image of the cursor. There are height*width **Words** in the cursor, or twice that many **Bytes**.

We will call the first byte beyond the image offset "ei" for "end of image."

mask	(+ei)	**Bytes**	The mask of the cursor. This is the same size as the image.

We will call the first byte beyond the mask offset "em" for "end of mask."

hotSpotY	(+em)	**Word**	The cursor's Y "hot spot."
hotSpotX	(+em+2)	**Word**	The cursor's X "hot spot."
flags	(+em+4)	**Flag Word**	Cursor flags: Bit 7: 1 = 640 Mode, 0 = 320 Mode All other bits are reserved and must be zero.
reserved	(+em+6)	**8 Bytes**	Reserved, must be zero.

The resource compiler template follows:

```
#define rCursor       $8027

/*---------------------- rCursor --------------------*/
type rCursor {
    height :
        hex integer;            /* height */
    width :
        hex integer;            /* width in words */
        hex string[2*$$Word(height)*$$Word(width)];    /* cursor image */
        hex string[2*$$Word(height)*$$Word(width)];    /* cursor mask */
        hex integer;            /* hotspot Y */
        hex integer;            /* hotspot X */
        hex integer;            /* flags */
        hex longint = 0;        /* reserved */
        hex longint = 0;        /* reserved */
    };
};
```

Following is a simple cursor example:

```
resource rCursor(1,fixed) {
        5,     /* height */
        2,     /* width */
        $"ffff0000"
        $"f00f0000"
        $"f00f0000"
        $"f00f0000"
        $"ffff0000",

        $"ffff0000"
        $"ffff0000"
        $"ffff0000"
        $"ffff0000"
        $"ffff0000",

        2,     /* hotspot Y */
        2,     /* hotspot X */
        $80    /* 640 mode */
};
```

Note that the resource is marked `fixed` so that its handle can be dereferenced and passed to `SetCursor`.

Version Resource (Type: $8029, rVersion)

Files may include a version resource with ID=1 for display by programs such as the Finder. All `rVersion` resource IDs other than 1 are **reserved** for future definition. The following describes the version resource format:

version	(+000)	**Long**	The application's version number, in Apple IIGS Long Version format. See Apple IIGS Technical Note #100, VersionVille, for more details.
country	(+004)	**Word**	An international country version code. Possible values are as follows:

verUS	0
verFrance	1
verBritain	2
verGermany	3
verItaly	4
verNetherlands	5
verBelgiumLux	6
verSweden	7
verSpain	8
verDenmark	9
verPortugal	10
verFrCanada	11
verNorway	12
verIsrael	13
verJapan	14
verAustralia	15
verArabia	16
verFinland	17
verFrSwiss	18
verGrSwiss	19
verGreece	20
verIceland	21

			verMalta 22 verCyprus 23 verTurkey 24 verYugoslavia 25 verIreland 50 verKorea 51 verChina 52 verTaiwan 53 verThailand 54
name	(+006)	**String**	Pascal string containing the desired name. May be the null string.
moreInfo	(+xxx)	**String**	Additional information to be displayed, such as a copyright notice. May be the null string. Recommended maximum length is about two lines of 35 characters each. May contain a carriage return (character $0D).

The resource compiler template follows:

```
#define rVersion      $8029

// Equates for the country code of an rVersion resource

#define Region                                          \
       verUS, verFrance, verBritain, verGermany,        \
       verItaly, verNetherlands, verBelgiumLux,         \
       verFrBelgiumLux = 6, verSweden, verSpain,        \
       verDenmark, verPortugal, verFrCanada, verNorway, \
       verIsrael, verJapan, verAustralia, verArabia,    \
       verArabic=16, verFinland, verFrSwiss, verGrSwiss,\
       verGreece, verIceland, verMalta, verCyprus,      \
       verTurkey, verYugoslavia, verYugoCroatian = 25,  \
       verIndia = 33, verIndiaHindi = 33, verPakistan,  \
       verLithuania = 41, verPoland, verHungary,        \
       verEstonia, verLatvia, verLapland, verFaeroeIsl, \
       verIran, verRussia, verIreland = 50, verKorea,   \
       verChina, verTaiwan, verThailand                 \

/*------------------- rVersion ------------------*/
type rVersion {
   ReverseBytes {
       hex byte;                                    // Major revision in BCD
       hex bitstring[4];                            // Minor vevision in BCD
       hex bitstring[4];                            // Bug version
       hex byte   development = 0x20,               // Release stage
                  alpha = 0x40,
                  beta = 0x60,
                  final = 0x80, /* or */ release = 0xA0;
       hex byte;                                    // Non-final release #
   };
   integer Region;                          // Region code
   pstring;                                         // Short version number
   pstring;                                         // Long version number
};
```

Following is a simple version example for “Super Graphics Destroyer”, version 2.0:

```
resource rVersion(1) {
       { $02,$0,$0,release,$00 },          /* version 2.0 release */
       verUS,                              /* US version */
       "Super Graphics Destroyer",         /* our app's name */
       "© 1991 Pretty as a Picture, Inc."  /* the copyright notice */
};
```

Note: For compatibility with the Finder, keep the `name` field identical across different versions of the same application, and make sure the `version` field increases on each later version released to your customers. The `moreInfo` field is not critical; if it changes between versions, it's no big deal.

Comment Resource (Type: $802A, rComment)

Files may include a comment resource with ID=1 for display by programs such as the Finder. All `rComment` resource IDs other than 1 are **reserved** for future definition. The following describes the comment resource format:

text	(+000)	**Bytes**	The comment. This is unformatted, 8-bit text suitable for displaying by a desktop program. No length limit is imposed by this resource format, although a practical limit of a few hundred characters is recommended.

The resource compiler template follows:

```
#define rComment      $802A

/*---------------------- rComment --------------------*/
type rComment {
    string;
};
```

Tagged Strings Resource (Type: $802E, rTaggedStrings)

A tagged strings resource lists pairs of Word values and Pascal strings.

count	(+000)	**Word**	Number of word/string pairs in this resource.
firstWord	(+002)	**Word**	Word value of first pair.
firstString	(+004)	**String**	Pascal string of first pair.
secondWord	(+xxx)	**Word**	Word value of second pair.
secondString	(+yyy)	**String**	Pascal string of second pair.
…			

The resource compiler template follows:

```
#define rTaggedStrings              $802E

/*------------------- rTaggedStrings --------------------*/

type rTaggedStrings {
              integer = $$Countof(StringArray);
              array StringArray {
                            hex integer;          /* Key integer */
                            pstring;                    /* String */
              };
};
```

Following is a simple `rTaggedStrings` example:

```
resource rTaggedStrings(1) {{
       $0050, "red",
       $0033, "green",
       $0100, "blue"
}};
```

Pattern List Resource (Type: $802F, rPatternList)

A pattern list resource contains zero of more 32-byte QuickDraw II patterns. (This resource type exists for your convenience. The System Software contains no direct support for resources of this type.)

firstPattern	(+000)	**32 Bytes**	First QuickDraw II pattern structure.
secondPattern	(+032)	**32 Bytes**	Second QuickDraw II pattern structure.
…			

The resource compiler template follows:

```
#define rPatternList         $802F

/*-------------------- rPatternList ---------------------*/

type rPatternList {
       array {
              array[32] {
                     hex byte;
              };
       };
};
```

Rectangle List Resource (Type: $C001, rRectList)

The rectangle list (type `rRectList`) is provided to allow an extensible, easily modifiable collection of QuickDraw II rectangle structures. This capability can enhance a developer's ability to modify on-screen displays without recompiling an entire application. This resource also enables easier cross-development and parallel development for the Apple IIGS and the Macintosh.

The following describes the rectangle list resource format:

count	(+000)	**Word**	Number of rectangles in this resource.
firstRectangle	(+002)	**8 Bytes**	First QuickDraw II rectangle structure.
…			Rectangles, eight bytes each.
lastRectangle	(+002+(8*(count-1)))	**8 Bytes**	Last QuickDraw II rectangle structure.

The resource compiler template follows:

```
#define rRectList    $C001
type rRectList {
        integer = $$Countof(RectArray);
        array RectArray {
            Rect;
            };
};
```

Print Record Resource (Type: $C002, rPrintRecord)

As a convenience for applications, a print record may be included as a resource of type $C002 (`rPrintRecord`). If more than one of these resources is present, the one to use as the document's primary print record is the first one. You can get this resource's ID by calling `GetIndResource` with type `rPrintRecord` and index 1. Storing the primary print record with ID = 1 is a good way to start.

Since the print record is filled in and interpreted by the printer driver, you can't always programmatically set options that are driver-specific. For example, although the ImageWriter driver stores the color-vs.-black and white option in one place, not all color printers will do the same thing. If you want to use driver-specific options on many printers, you can use those printer drivers to create print record that reflect the options you want and store those records as resources. Then, if you need a pre-initialized print record with the options you want, you may already have one.

print record	(+000)	**160 Bytes**	The print record. The handle to this resource is suitable for passing to any Print Manager call that requires a print record handle.

Since applications shouldn't create print records from scratch, but rather allow printer drivers to fill them in with `PrDefault` and `PrVerify`, the resource compiler template that follows only allocates 160 bytes for storage.

```
#define rPrintRecord  $C002

/*-------------------- rPrintRecord ------------------*/
type rPrintRecord {
       array[160] {
              hex byte;
       };
};
```

Font Resource (Type: $C003, rFont)

Some applications wish to keep fonts with the application itself instead of in a separate font file. This isn't always advisable—fonts not in font files can't be easily used in other applications, which may confuse users who, for example, use text-editing desk accessories and can't get at certain fonts except in certain applications. Also, fonts not in the Fonts directory must be completely memory-resident where normal Fonts are only loaded from disk when they're needed.

Nevertheless, in some cases keeping fonts inside an application file is necessary or desirable. In such cases, the `rFont` resource is a convenient way to keep a font around. The resource is a QuickDraw II Font, as defined in *Apple IIGS Toolbox Reference*, Volume 2. The font family name is the resource's name—the same Pascal string that normally precedes the QuickDraw II font record in the font file.

font	(+000)	**Bytes**	The font record, without the font family name.

Since the font record is a bunch of variable-sized tables, and since you probably want to use a font editor (and not the resource compiler) to create fonts, the following resource compiler template isn't very revealing.

```
/*-------------------- rFont ------------------*/

type rFont {
       hex string;
};
```

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1–3
- Apple IIGS Technical Note #100, VersionVille
- HyperCard IIGS Technical Note #3, Pitching Sampled Sounds

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#77: Print Manager & AppleTalk Configuration Files

Written by: Jim Luther January 1990

This Technical Note describes the Print Manager user configuration file Printer.Setup and the AppleTalk user configuration file ATInit. This Note also describes a limitation of the Print Manager call `PrGetUserName`, which is a result of the way configuration information is stored in the Printer.Setup file.

Printer.Setup and ATInit

What Are the Printer.Setup and ATInit Files?

The Print Manager user configuration file Printer.Setup, which is found in the System:Drivers directory of the Apple IIGS boot disk, is used by the Print Manager tool set to keep the name of the printer driver and port driver you've selected between system boots. In addition, if you've selected a network printer, Printer.Setup contains the selected printer's network address and your machine's User Name. The file format of Printer.Setup has not been published because revisions have been made, and may be made again, to the Apple IIGS System Software, which can change Printer.Setup's file format.

The AppleTalk user configuration file ATInit, which is found in the System:System.Setup directory of the Apple IIGS boot disk, is used to keep the default AppleShare startup application, the default AppleShare prefix, the default AppleTalk User Name, and the default AppleTalk printer entity name (the network printer entity used by AppleTalk's Remote Print Manager) between system boots. The file format of the ATInit file was published incorrectly in the *AppleShare Programmer's Guide for the Apple IIGS*. The correct file format for ATInit will be discussed later in this Note.

It is important to remember that the Print Manager tool set uses the information from the Printer.Setup file only, and that AppleTalk and AppleShare use the information contained in the ATInit file only. It is also important to note that the Print Manger tool set, which is used to print QuickDraw II graphics, and AppleTalk's Remote Print Manager (RPM), which is used to print ASCII data to network printers, are not the same thing even though both contain the words "Print Manager."

What Writes to the Printer.Setup and ATInit Files?

Before Apple IIGS System Software 5.0, Printer.Setup and ATInit were handled as completely separate configuration files. The Print Manager call `PrChoosePrinter` allowed you to select the printer and port drivers the Print Manager would use and wrote the printer and port driver selections to the Printer.Setup file. The AppleTalk application Chooser.II let you select the printer AppleTalk's Remote Print Manager would use and wrote the printer entity selection to the ATInit file.

With System Software 5.0 all printer selections for **both** the Print Manager and AppleTalk are made by using one of the Control Panel NDA's printer CDevs. All printer CDevs (e.g., DirectConnect, ATIWriter, ATLQIWriter, and ATLWriter) write the new printer and port driver selections to the Printer.Setup file. However, if the printer selected uses the AppleTalk port (i.e., the selection is made with the ATIWriter, ATLQIWriter, or ATLWriter printer CDevs), then the selected printer's network address and your User Name are written to both the Printer.Setup and the ATInit files. The DirectConnect CDev does not write any information to the ATInit file. If AppleShare is installed, then the AppleShare CDev will also write your User Name to the ATInit file.

On AppleShare file servers with the Apple II Setup option installed, the ATInit file in User folders will also be written to by the AppleShare Admin application when the Apple II startup information is set.

When are the Printer.Setup and ATInit Files Read?

The Printer.Setup file is read by the Print Manager and by the printer CDevs. The Print Manager reads the information contained in the Printer.Setup file whenever the Print Manager needs to load a printer driver or a port driver into memory. A printer CDev reads the information contained in the Printer.Setup file when that CDev is selected so it can know the current printer and port selections.

Ways the printer driver and the port driver might be unloaded and need to be loaded (which will cause Printer.Setup to be read by the Print Manager) are as follows:

- The Print Manager is shut down.
- The current printer driver or port driver is changed with a Control Panel printer CDev. When a new printer or port is selected with a printer CDev, the current drivers are unloaded from memory so the Print Manager will be forced to read the new printer and port selections from Printer.Setup.
- Your application makes the `PMUnloadDriver` Print Manager call.

An application can load one or both of the drivers (which will cause Printer.Setup to be read by the Print Manager) by making the `PMLoadDriver` call. The AppleTalk user configuration information contained in the ATInit file is read during system startup as part of AppleTalk's initialization.

Network Booting and Printer.Setup

When Apple IIGS computers are booted over an AppleShare network, they all share a single copy of the Printer.Setup file. That means all machines must use the same printer and port driver selections that are stored in the Printer.Setup file. If all machines are expected to be able to print using the Print Manager tool set, then the printer and port selection stored in Printer.Setup must be something that **all** can use. The only two options are:

- A single shared network printer for all machines (i.e., a LaserWriter, an AppleTalk ImageWriter, or an AppleTalk ImageWriter LQ). In situations where many machines are booted over a single file server, this may cause the workload on the single shared printer to be unacceptable.

- A direct-connect printer on each machine. The limitations of this solution are that the printers must be of the same type (all ImageWriters, all ImageWriter LQs, or all Epsons) and all machines must use the same printer port (either printer or modem).

The server administrator should set the default printer selection, which will be used by all machines, by using one of the Control Panel NDA's printer CDevs. Then, the access privileges to the server's System:Drivers directory should be set to "Bulletin Board" (i.e., Everyone See Folders, Everyone See Files, Owner Make Changes) so other machines cannot change the printer and port selection.

Using User Names

The User Name We Use

You may have noticed that you see your AppleTalk User Name in the Control Panel's AppleShare and printer CDevs. AppleShare allows a machine's User Name to be up to 31 characters long. The CDevs read the User Name from the ATInit file. The AppleShare and printer CDevs also store the complete User Name back into the ATInit file.

PrGetUserName (Almost)

The Printer.Setup file sets aside 15 characters for the User Name so the printer CDevs store only the first 15 characters of the User Name in the Printer.Setup file. This limitation is leftover from early Print Manager implementations of the `PrChoosePrinter` call, which limited the User Name length to 15 characters.

Since the Print Manager gets the User Name it uses from the Printer.Setup file, the User Name returned by the Print Manager call `PrGetUserName` will be truncated to 15 characters if the complete AppleTalk User Name is 16 characters or longer.

Where to Get the Complete User Name

If your application needs the complete default AppleTalk User Name, it can be read from the ATInit file. When an Apple IIGS is booted from a local disk volume that has AppleShare or at least one of the AppleTalk network printers installed, ATInit will be found in the System:System.Setup directory of the local boot volume. When an Apple IIGS is booted over AppleTalk, ATInit will be found in the Users:YourName:Setup directory of the AppleShare boot volume (where YourName is the User Name used to log on to the boot server).

The ATInit File Format

The *AppleShare Programmer's Guide for the Apple IIGS* shows the file format of the ATInit file as it is stored on an AppleShare boot volume. However, the file format of ATInit is not always as shown in that manual. In all cases, ATInit will contain the three required data fields `UserName`, `PrinterFlags`, and `PrinterTuple` at the end of the file. Before those data fields, ATInit may also contain executable code or additional data fields. Since the three required data fields are directly before ATInit's end-of-file (EOF), you can find them relative to ATInit's EOF using the displacements listed in Table 1.

Displacement to ATInit EOF	Size	Field Name	Description
133	33 Bytes	`UserName`	A Pascal-type string containing the default User Name. It consists of a length byte followed by up to 31 bytes of ASCII data and a single, unused byte. This field is always 33 bytes long.
100	Byte	`PrinterFlags`	This is the Flags field used by the Remote Print Manager's default network printer.
99	99 Bytes	`PrinterTuple`	This field specifies the name of the default network printer used by the Remote Print Manager. The `PrinterTuple` field is in standard Name Binding Protocol (NBP) format. This field is always 99 bytes long.

Table 1–Offsets of Required Data Fields

If the ATInit file is on an AppleShare server, it will have 6 additional data fields (`PathVolID`, `PathDirID`, `Path`, `PrefixVolID`, `PrefixDirID`, and `Prefix`) directly before the three required data fields. These fields can also be found relative to ATInit's EOF using the displacements listed in Table 2.

Displacement to ATInit EOF	Size	Field Name	Description
275	Word	`PathVolID`	The Volume ID number of the user's AppleTalk startup application.
273	Long	`PathDirID`	The Directory ID number of the user's AppleTalk startup application.
269	65 Bytes	`Path`	The Pathname of the user's AppleTalk startup application.
204	Word	`PrefixVolID`	The Volume ID number of the user's AppleTalk default prefix.
202	Long	`PrefixDirID`	The Directory ID number of the user's AppleTalk default prefix.
198	65 Bytes	`Prefix`	The user's AppleTalk default prefix.

Table 2–Offsets of Optional Data Fields

The displacements in Tables 1 and 2 can be used with the GS/OS `SetMark` call to move the file mark to the beginning of any of the above fields. The `SetMark` call's `base` field should be set to $0001 so the mark will be set equal to EOF minus the displacement.

Further Reference

- *Apple IIGS Toolbox Reference*
- *Inside AppleTalk*
- *AppleShare Programmer's Guide for the Apple IIGS*\

Apple IIGS
#78: Bank Alignment and Memory Management

Revised by: Matt Deatherage May 1992
Written by: Matt Deatherage March 1990

This Technical Note discusses the way the Memory Manager deals with requests for memory that is already in use, and why this can be really annoying.
Changes since March 1990: Included new information about some smarter algorithms in System Software 6.0 and later which can avoid problems some of the time.

The Memory Manager is a sophisticated software module that provides the framework for the allocation, moving, management, and disposal of blocks of memory; however, it's not magic.

When you ask the Memory Manager for a block of memory and it's not immediately allocatable, the Memory Manager starts through the procedure for purging, compacting, and calling out-of-memory (OOM) queue routines until, at the end of its rope, it finally gives up and returns error $0201. The exact procedure is repeated below, taken from Volume 3 of the *Apple IIGS Toolbox Reference*. Note that each successive step is only taken if, after the previous step, the requested memory still isn't available.

1. Calls each OOM queue routine until either all routines have been called or until one OOM queue routine reports that it has freed enough memory to satisfy the request.
2. Compacts memory.
3. Purges all level 3 handles. If this frees enough memory, compaction occurs.
4. Purges all level 2 handles. If this frees enough memory, compaction occurs.
5. Purges all level 1 handles. If this frees enough memory, compaction occurs.
6. Calls each OOM queue routine again until all have been called or until one OOM queue routine reports that it has freed enough memory to satisfy the request.
7. Gives up and returns error $0201.

This strategy works pretty well—as long as the request is for a block of memory wherever it fits. If someone has asked the Memory Manager for memory at a specific address, things get stickier.

Suppose that you've asked the Memory Manager for a handle starting at the beginning of bank 2, and that something else (i.e., the ProDOS FST) is already using that memory. The Memory Manager notices that the handle isn't immediately available, so it starts going through the listed procedures. Since the handle for the ProDOS FST is neither purgeable nor movable and GS/OS isn't likely to give it up in an OOM queue routine, the request fails and the Memory Manager returns error $0201.

However, the Memory Manager went through **all** the steps listed to get to the seventh step, the error. The Memory Manager has no way to know that one of the OOM queue routines isn't going to give up that particular handle and allow the request to be fulfilled. The OOM queue routines

cannot know themselves, since they are only told how much memory is needed, not where it has to be. Therefore, whenever the Memory Manager returns error $0201, **all** purgeable handles have been purged.

This is particularly annoying to loaders. OMF supports a "bank-aligned" attribute for load segments, and the loaders ensure that such segments are loaded at the beginning of some bank or another. The Memory Manager does not have a "bank-aligned" attribute for handles, so the loaders have to do these things themselves. They do this by asking for a handle of the appropriate size at the beginning of bank two. If this fails, the loaders try again with bank three, then bank four, and so on through the end of memory.

Since some part of GS/OS is almost always occupying the memory at the beginning of bank two, which is where the loader first attempts to load a bank-aligned segment, the presence of such a segment in a load file virtually guarantees that all purgeable handles are purged when the file is loaded. This kicks out dormant applications and zombie-state tool sets, among other things, requiring they be loaded from disk again when needed.

Starting with System Software 6.0, the Loader attempts an alternate strategy first—it tries to allocate an entire bank of memory that is page aligned and doesn't cross a bank boundary. This block, if available, will by definition be bank-aligned. Since code segments can't be larger than 64K, such a block can always hold a bank-aligned segment. The Loader now tries to allocate such a block and if it finds one, it immediately disposes of it and allocates a block of the right size at the same bank address.

If this strategy succeeds, it's a lot faster than the other method and may avoid purging all of memory. If it fails, though, the Memory Manager still goes through all seven steps before returning error $0201, so all of memory may still be purged. It's just less likely in System Software 6.0 and later.

It doesn't make sense to bank-align a small segment, and small segments fit better into fragmented memory. If you use large segments anyway, consider the trade-off: bank-aligning a segment may purge memory at load time, but your linker may be able to generate smaller OMF, decreasing disk size and load time.

Summary

The general recommendation against asking for specific blocks of memory is well-known to most developers; the reasons outlined above simply add fuel to the fire against such programming practices. What isn't as widely known is that having a bank-aligned load segment in a load file may cause everything purgeable to be purged, and could also cause OOM queue routines to dispose of handles when there really isn't any kind of memory shortage.

Apple advises developers to carefully consider the advantages and disadvantages of bank-aligned segments before including one in a load file.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1 and 3
- *GS/OS Reference*

Apple IIGS
#79: Integer Math Data Types

Revised by: Jim Luther May 1990
Written by: Dan Strnad March 1990

This Technical Note describes the format of `Fixed` and `Frac` data types used by the Integer Math tool set and operations performed on the Integer Math numerical data types.
Revised since March 1990: Fixed original date, bit numbering of diagrams, and a multiplication sign in the equation.

As stated in Volume 1 of the *Apple IIGS Technical Reference*, the Integer Math tool set provides the following numerical data types:

```
Integers
Longints
Fixed
Frac
Extended
```

The precise format of the `Fixed` and `Frac` data types is not provided in the reference manual, so this Note details these formats.

The format for the `Fixed` data type is stated in the manual as being a 32-bit signed value with 16 bits of fraction. This means that the low-order 16 bits of the `Fixed` format data value are considered as a fraction of 2^16, which is the binary number represented by a one followed by 16 zeroes ($10000). In other words, a `Fixed` value is the same as a long integer value whose binary point has been moved to the left 16 places. In this representation, if the low-order part of the `Fixed` format data value were $8000, the fractional value would be equal to 1/2. A low-order part of $C000 would represent a fractional part equal to 3/4. Therefore the highest value that a `Fixed` can contain is 32,767 and 65,535/65,536; the least value is equal to -32768.

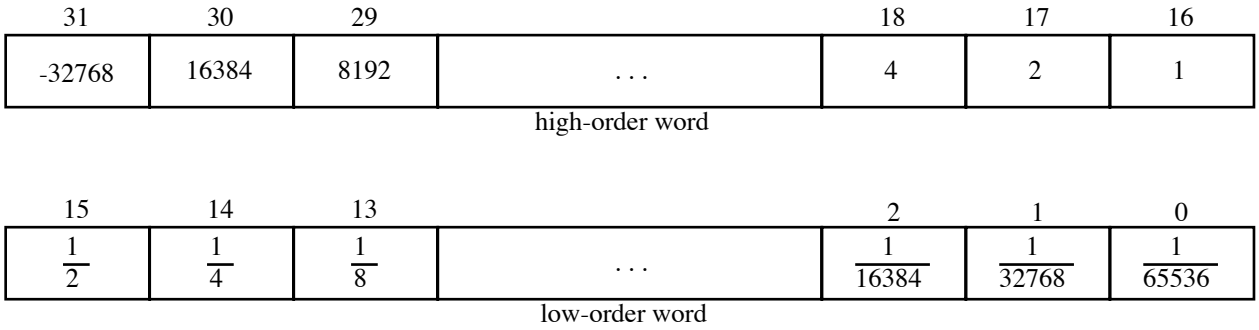

Figure 1–Fixed Data Type

The format for the `Frac` data type is stated in the manual as being a 32-bit signed value with 30 bits of fraction. This means that the low-order 30 bits of the `Frac` format data value are considered as a fraction of 2^30, which is the binary number represented by a one followed by 30 zeroes ($40000000). In other words, a `Frac` value is the same as a long integer value whose binary point has been moved to the left 30 places. The high-order 2 bits of the `Frac` format data value are treated as follows. The high bit has a value of -2 and the low bit has a value of 1. Therefore the highest value that a `Frac` can contain is 1 and ((2^30)-1)/2^30; the least value is equal to -2.

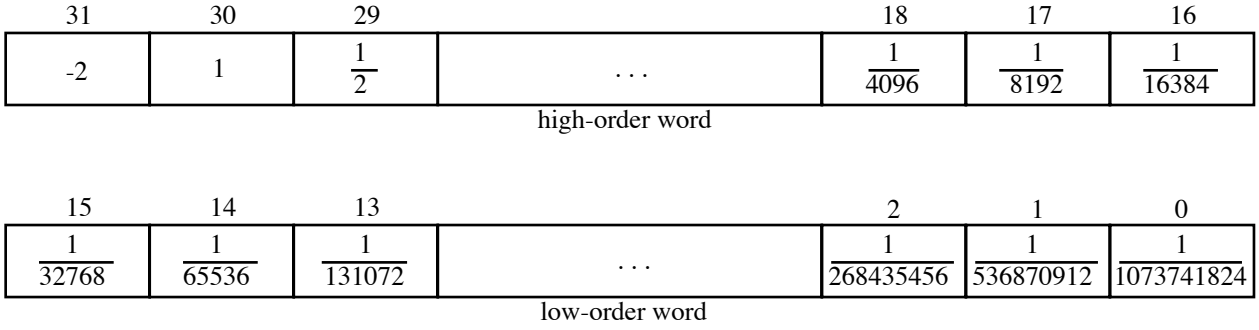

Figure 2–Frac Data Type

Note that for `Longints`, `Fixed`, and `Frac` values, the hex representations of the largest and smallest data values are $7FFFFFFF and $80000000, respectively.

A property of the `Fixed` and `Frac` data types is that two `Fixed` or two `Frac` values may be added or subtracted just as if they were 32-bit integers. To demonstrate this, imagine scaling the numbers by a given factor to make them integers. After adding the numbers, the sum could be scaled back down by the same factor. This follows from the distributive property of multiplication over addition, which allows one to make the inference shown in the equations which follow. In these equations, `V1` and `V2` are both either `Fixed` or `Frac` values. The value for `C` being discussed, which illustrates the ability to scale `Fixed` and `Frac` values, is 2^16 for `Fixed` values of `V1` and `V2`, or 2^30 for `Frac` values of `V1` and `V2`.

$$\frac{(C * V1) + (C * V2)}{C} = \frac{C * (V1 + V2)}{C} = V1 + V2$$

Similarly, two `Fixed` or two `Frac` values may be compared, as `Longints` are compared, with one another. In general, the comparison, addition, and subtraction operations used for long integers may also be performed on any two `Fixed` or any two `Frac` values.

Further Reference

- *Apple IIGS Technical Reference Manual*
- *Apple Numerics Manual, Second Edition*