Apple II
Technical Notes

Developer Technical Support

Apple IIGS #92: Twisted Tales of TextEdit

Revised by: Dave Lyons December 1991
Written by: C.K. Haun <TR> and Dave Lyons September 1990

This Technical Note discusses some undocumented features and some bugs in the TextEdit tool set through System Software 5.0.4.
Changes since November 1990: Noted that a non-control `TENew` creates a Text Edit record for the current port.

TENew

TextEdit records you create with `TENew` are always tied to the current port at the time of the `TENew` call, whether or not the `fNotControl` bit is set. (For TextEdit controls, `NewControl2` is the preferred call.)

TEInsert

Using the `TEInsert` call on an invisible TextEdit record causes the screen to scroll, exactly as if the TextEdit record were visible.

If you use `LETextBox2` style text as input for a `TEInsert` call, any style change information contained at the end of the `LETextBox2` text is ignored. To ensure that the style change is not ignored, append an additional character at the end of the block, then delete (with `TESetSelect` and `TEDelete`) the extra character after the `TEInsert` call.

TEGetText

The documentation for `TEGetText` says that a `dataFormat` value of $4 returns the text as “Formatted for input to LineEdit `LETextBox2`”. This is not a reliable return method—this call may or may not succeed. Greater chance for success occurs with less than 4,000 characters in the TextEdit record.

`TEGetText` also supports getting just the text of the current selection range. Adding `$0020` (`onlyGetSelection`) to the number passed in `bufferDescriptor` returns the text of the current selection. This technique does **not** work with data format `LETextBox2`, but does work with all other formats. Also, there is no corresponding bit for the associated style record, so you cannot get the style for just the current selection this way, if you request style information you get a `styleRef` for the entire TextEdit record.

TEClick

Using `TEClick` or `TestControl` on an inactive record currently causes that record to activate.

TERuler

Pixel tabbing values must all be greater than zero or TextEdit loops infinitely on a tab.

TEGetRuler & TESetRuler

`TERuler`, for the default ruler or any ruler that uses a `tabType` value of $1 returns a ruler four bytes longer than described in the documentation. The extra four bytes are all $FF, and they are the terminator characters for `tabType` $2 rulers. Expand your buffers by four bytes to prevent overwriting any data. TextEdit also expects the additional information on a `TESetRuler` call, so you should pad your ruler with four $FF bytes if you are using a type $1 ruler.

TESetText

Passing a zero-length class one input string (a word length string with the word set to zero) to `TESetText` causes TextEdit to crash.

TEPaintText

`TEPaintText` currently prints colored text in only four colors.

It's Not Dirty, It's Text

There has been some confusion about determining if a TextEdit record has been changed. The documentation has been a little vague, and the process itself has mislead some people. Here is The Truth: there **is** a TextEdit dirty flag, and you can use it and rely on it to tell you when a TextEdit record has changed.

The TextEdit dirty flag is bit 6 (`fRecordDirty` in the E16.TextEdit interface file) of the `ctlFlag` byte. This has caused some confusion because the `ctlFlag` byte is at offset `$12` in the control definition **template**, and it is at offset `$10` in the TextEdit or Control **record**. Just remember that it is **not** in the same place in the record as it is in the template.

If it is set, then the TextEdit or Control record has been changed since the last time the dirty bit was cleared. The dirty bit is clear initially when you create the TextEdit or Control record. Anytime after that, if the user enters text into the TextEdit record, TextEdit sets the dirty flag. It is up to your application to clear the dirty flag; TextEdit has no way of knowing when you’ve saved or cleared data.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 3

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#93: Compatible Printing

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

This Technical Note discusses printing on the Apple IIGS and how you can make your printing code more compatible.
Changes since September 1990: Added a note about expecting print records to keep the same attributes across Print Manager calls. Added the StyleWriter's `iDev` value.

How Does Printing Work Anyway?

There are, in general, two types of printing done on the Apple IIGS. The first kind is "desktop" printing, which uses the Apple IIGS Print Manager to render images created by QuickDraw II onto an output device. The other kind of printing is "text" printing, which is similar to the way classic Apple II applications print—you send ASCII text somewhere and a printer prints it as ASCII text. This printing normally involves no graphics and is very quick.

This Note covers both types of printing, and by understanding the internals and the methods used to print, you can avoid compatibility headaches in the future.

Desktop Printing

Desktop printing uses the Apple IIGS Print Manager. The process is described in detail in the Print Manager chapter of the *Apple IIGS Toolbox Reference*, and usually consists of a simple print loop:

```
Open a document (PrOpenDoc)
        Open a page (PrOpenPage)
        Draw or Image the page in your favorite way
        Close the page (PrClosePage)
        Repeat for each page
Close the document (PrCloseDoc)
Print the document if it's spooled (PrPicFile)
```

Note that you should **always** call `PrPicFile` at the end of your print loop. It completes the printing process, even for immediate or draft printing.

There's one real secret about the Print Manager that can cloud your understanding of printing—the Print Manager doesn't actually do anything. It loads, unloads, and keeps track of printer drivers and port drivers and performs some necessary housekeeping, but that's about it. Many people believe that the Print Manager is responsible for all imaging, managing documents, managing a printing `grafPort` and such, but it's not. (The myth is perpetuated by the *Toolbox Reference* which refers to these functions as handled by the Print Manager.) In fact, these functions are handled by printer drivers.

You actually call the printer driver for all of the routines in the print loop; all the Print Manager does is make sure the driver is loaded and dispatch to it. Therefore, most of the compatibility issues you have with printing are not with the Print Manager, but with printer drivers.

Dealing With the Print Record

It's the printer driver's job to get information about a printing job from the user (it's the printer driver that handles the style and job dialog boxes, since the Print Manager cannot generically know what style and job options any printer can support), keep track of it, and print the document using those settings. Those settings are kept in a data structure associated with a document known as a print record.

Apple had only released two printer drivers at the time the first volume of the *Toolbox Reference* was published, and therefore the descriptions of the print record in that volume tend to be absolute. For example, the `iDev` field is documented as "one for an ImageWriter and three for a LaserWriter." In fact, the `iDev` field is the only method of print record interpretation available and there are several values for it:

$0001 = ImageWriter
$0002 = ImageWriter LQ
$0003 = LaserWriter
$0004 = Epson
$0065 = StyleWriter
$8001 = Generic dot-matrix (interprets the style subrecord like the ImageWriter driver)
$8003 = Generic laser printer (interprets the style subrecord like the LaserWriter driver)

If you have checks in your code like "If it's not $0001, it must be a LaserWriter," you have problems with most of the other printer types.

The $8000 and greater `iDev` values are defined for third-party printer drivers. The printer driver has no way other than the print record to keep track of values for a given print job, so it has to store all such information in the print record. If all third-party drivers use proprietary style subrecord formats, no applications can read or set any of those values. Those drivers which can use the compatible $8000 and greater `iDev` values indicate to applications that the definitions in *Toolbox Reference* for the ImageWriter and LaserWriter drivers apply to these drivers as well. `iDev` values of $0002 or $0004 also interpret the style subrecord as the ImageWriter driver does.

Print Record Rules

Remember: the print record is the only way the printer driver has to maintain information about a particular job. The print record belongs to the user, the document, and the printer driver—**not** the application. Here are some rules for staying out of print record trouble.

- Always call `PrValidate` when changing fields in the print record. Even if a driver interprets the style subrecord like the ImageWriter driver, it may not support all the ImageWriter's style features (e.g., color printing). Calling `PrValidate` every time you change something in the print record gives the printer driver a chance to look at the havoc you've wreaked and correct it if necessary.

  You do not always get a feature you want. If a printer does not support color printing, you can set the "color" bit all day long and `PrValidate` clears it every time. You should be prepared for a new printer driver that does not support the features you want, and inform the user that the feature is not supported by this printer.

- Do not patch `PrValidate` to make it ignore bogus values in the print record unless instructed to do so by the printer driver author.

- **Never**, **never** tread on reserved fields in the print record. If you find a particular driver storing useful values some place, forget it. This is the only place a driver has to store information about a print job and some of it is not going to be supported.

  In particular, never try to interpret any values you may find in the `printX` subrecord of the print record. This subrecord is for the private use of printer drivers. Although `printX` is currently the worst compatibility risk, you must not tamper with other reserved fields.

- Don't assume that the print record will keep the same memory attributes across calls to the Print Manager (and therefore the printer driver). Specifically, don't assume that a print record will stay locked across calls to the Print Manager.

- If you want to learn more about printing, learn how printer drivers work. The specifications are in Apple IIGS Technical Note #35, appropriately entitled "Printer Driver Specifications." An understanding of how printer drivers do their work is an understanding of how printing works.

Text Printing

Text printing generally uses the built-in ASCII mode of most dot-matrix printers to print text quickly and efficiently.

Desktop printer drivers often have a "draft" mode, where they print text immediately instead of imaging it in the appropriate font and style. This is accomplished by intercepting low-level QuickDraw II routines called bottleneck procedures. When QuickDraw is called to draw text, the printer driver gets control instead and sends the text to the printer.

Although this is useful to users of desktop printer drivers, it is not a required feature of any printer driver, and those that do implement it each do so in their individual way. For example, the LaserWriter driver doesn't support this model of "draft" printing because the LaserWriter is normally a PostScript® device—sending straight ASCII to it doesn't necessarily work.

To imitate the way classic Apple II applications print, your application prompts the user for some device through which to print, and ASCII characters are sent through that device. There are a few ways to do this.

Using the Print Manager

You can still use the Print Manager to print in ASCII mode by bypassing the printer driver. Simply use the Port Driver to send ASCII characters to the given target device with the `PrDevWrite` call. The specifications for Port Driver calls are in Apple IIGS Technical Note #36, also appropriately entitled "Port Driver Specifications." You make port driver calls as if they were Print Manager calls.

Although this method has been used, Apple does not recommend it. If the selected port driver is a network driver, this method is troublesome.

Using the Text Tools

By using the Apple IIGS Text Tools, you can ask the user what slot to print through and send ASCII characters to that slot or port. Although this is better than using the Port Driver, it still has problems. The Text Tools cannot be fully GS/OS Slot Arbiter compatible; therefore, there might be GS/OS devices accessible to the user to which your application does not let him print. Also, it’s difficult to detect which slots really have Text Tools’ devices without knowing about Apple II firmware, and prompting the user for a slot number invites trying to print to the disk firwmare, which usually justs reboot the machine (unceremoniously).

Using GS/OS

GS/OS supports character drivers, such as printer interfaces, and using them is the best way to handle ASCII printing. GS/OS supports loaded drivers for character devices if you have them, and generates drivers for character devices it can recognize. In addition, GS/OS drivers have identification words so you can prompt with real messages instead of cryptic slot numbers.

You can use the GS/OS call `DInfo` to loop through all drivers and prepare a list of character drivers. You can then change their device IDs into text phrases, place them in a list, and prompt the user to select one. This call usually results in a list such as “Printer port, Modem port, Remote Print Manager, Printer interface, Text screen [the Console driver].” You may wish to change the names of the devices slightly to make the choice easier (e.g., “network printer” instead of “Remote Print Manager”).

Apple strongly recommends using GS/OS for ASCII printing from 16-bit applications.

Note: The Remote Print Manager (.RPM) device driver in System Software 5.0 to 5.0.2 has a bug which causes character loss. System Software 5.0.3 fixes this bug.

Further Reference

- *Apple IIGS Toolbox Reference*
- *GS/OS Reference*
- Apple IIGS Technical Note #34, Low-level QuickDraw II Routines
- Apple IIGS Technical Note #35, Printer Driver Specifications
- Apple IIGS Technical Note #36, Port Driver Specifications
- Apple IIGS Technical Note #69, The Ins and Outs of Slot Arbitration
- Apple IIGS Technical Note #75, BeginUpdate Anomaly

Developer Technical Support

Apple IIGS
#94: Packing It In (and Out)

Revised by: Dave Lyons May 1992
Written by: C.K. Haun <TR> September 1990

This Technical Note discusses a potential problem with the Miscellaneous Tools routine `UnPackBytes`.
Changes since September 1990: Noted that the problem detecting the end of the unpack-to buffer near the end of a bank is fixed in System 6.0.

`PackBytes` and `UnPackBytes` are handy data compression and expansion routines built into the Apple IIGS System Software. Using them can dramatically reduce the amount of space your application uses on disk or in memory, but you need to understand how these calls work to avoid problems in your applications.

Buffer Size, Buffer Size, Buff, Buff, Buffer Size

There are some situations where the Miscellaneous Tools call `UnPackBytes` does not function as expected and can cause your application to loop infinitely while you're waiting for an unpacking process to finish.

The following packed data and code (in APW assembly) demonstrates the problem. It shows a small routine that unpacks data in two steps, simulating the situation in many applications where an arbitrary amount of data is unpacked in a variable amount of unpacking actions, depending on the results of the last unpack pass.

```
UnPackBuffer      ds    160                    ; area to unpack the data to
UnPackBufferPtr   dc    i4'UnPackBuffer'       ; pointer to unpacking buffer
UnPackBufferSize  ds    2
temp              ds    2

PackedData        dc h'FFFFFFFF'
EndPackData       anop
PackLength        dc i2'EndPackData-PackedData' ; how many bytes of packed data

* In packbytes format $FFFF means '64 repeats of the next byte ($FF) taken as 4 bytes' as
* described on page 14-39 of Toolbox Reference, so
* this data should unpack into 512 $FF bytes
* The following code loops infinitely

                  lda   #160                 ; Unpack buffer size
                  sta   UnPackBufferSize
UnPackLoop        pea   0                    ; return space
                  pushlong #PackedData       ; pointer to packed data
                  pea   2                    ; size of the packed data, unpack two bytes
                                             ; at a time
                  pushlong #UnPackBufferPtr  ; pointer to pointer to unpacking buffer
                  pushlong #UnPackBufferSize ; pointer to word with the size of the
                                             ;  unpacking buffer
                  _UnPackBytes
                  pla                        ; returns 0 bytes unpacked
                  sta   temp
                  lda   PackLength
                  sec
                  sbc   temp                 ; subtracting it from our known
                  sta   PackLength           ; length of packed data
                  bne   UnPackLoop           ; this is always be non-zero
```

The problem is in the data and the buffer size. `UnPackBytes` is being told to unpack two bytes ($FFFF), which generate 256 bytes of unpacked data, into a 160-byte buffer. Instead of reporting an error with this condition, `UnPackBytes` instead just does nothing and passes back zero as the returned number of bytes unpacked. If you are relying on the unpacked byte count returned to control your unpacking loop, then you may encounter this problem.

`UnPackBytes` can be used to unpack in multiple steps, of course, but it cannot unpack a partial record. It cannot unpack 160 bytes of the 256 bytes specified in this record because `UnPackBytes` does not maintain any state information, so it must unpack full records or do nothing. If the buffer had been 256 bytes, this call would have succeeded.

The Fix

Fortunately, it's easy to avoid this situation if you know that it can exist. Simply, always supply `UnPackBytes` with a buffer that is big enough for it to unpack at least two bytes (a flag or count byte and a data byte). The largest value of a flag or count word possible is $FF, 64 repeats of the next byte taken as four bytes, which generates 256 unpacked bytes. So always give `UnPackBytes` a 256-byte long output buffer and you should never encounter this problem.

Check Your Current Applications

Please check your current applications to see if you could encounter this problem. One of the most likely places for this error to occur is in applications that process Apple Preferred (file type $C0, auxiliary type $0002) pictures. While most pictures currently available are screen-width or less (160 bytes or less per scan line), the Apple Preferred format and QuickDraw II both support pictures that are wider than the current Apple IIGS screen. If someone has created a picture with a `PixelsPerScanLine` value of 1,280 with a `ModeWord` of $0080, it would generate a scan line that was 320 bytes long. If a scan line in this hypothetical picture were all white, for example, the first two bytes of the packed scan line would be $FFFF, and applications that assume a standard maximum 160 bytes per scan line would not handle this correctly.

But That's Not All…

In System Software earlier than 6.0, `UnPackBytes` has some other buffering problems of which you need to be aware. The size and location of the input buffer (the buffer containing your packed data) can also cause problems.

You can ignore this section if your application requires System 6.0.

Note: These problems only occur if you are doing multipass unpacks. If you always unpack a packed data range in one pass (with one call to `UnPackBytes` for the whole data set) then you are not affected by these problems, and the restrictions described herein do not apply.

Multipass Restrictions

When performing a multipass unpack (as described on pp. 14-43..44 of the *Apple IIGS Toolbox Reference*, Volume 1) the packed data needs to follow two rules.

Rule 1: Your packed data buffer cannot cross a bank boundary.
Rule 2: Your packed data buffer needs to be at least 65 bytes longer than the actual size of the data.

These rules are required by a bug in `UnPackBytes`. When `UnPackBytes` begins to unpack a record, it checks the record data to see if there are enough bytes in the current source buffer to unpack the number of bytes requested in the record header (described on pg. 14-39 of the *Apple IIGS Toolbox Reference*, Volume 1). If there are not enough bytes left for the current record (i.e., the header says to process 63 bytes, and there are only 30 left in the buffer), `UnPackBytes` returns to the caller. The caller then adjusts the source buffer for the next pass based on the amount of actual bytes unpacked, so the bytes left over from the last pass get processed the next time.

The problem occurs when the partial record is close to the end of a bank. When `UnPackBytes` checks to see if there is enough data left in the buffer, the check is flawed when the real end of the buffer is near the end of a bank, and a complete copy of the partial record would extend into the next bank. `UnPackBytes` erroneously thinks that the record is complete, and happily unpacks the remaining actual packed data, plus random information from the next bank. It continues to unpack nonsense data until it fills the unpacking buffer and the number of bytes unpacked returned by the `UnPackBytes` call is greater than the `bufferSize` parameter passed as input.

To prevent this bug from occurring, you need to make sure that the buffer for the packed data is at least one record length away from the end of a memory bank. Since the largest packed data record is one flag byte and 64 data bytes, adding 65 bytes to the end of your buffer does the trick. This ensures that your packed data is 65 bytes away from the end.

Following is an example of a safe way to prepare your packed data buffer for multipass unpacking, in APW assembly:

```
* Some data space
myCallBlock  dc     i2'2'               ; two parameters
fileRefNum   ds     2                   ; file reference number
EOFreturned  ds     4                   ; file length returned by this call
myIDNumber   ds     2                   ; your application memory manager ID number
* assume that a packed data file is open, and it's a plain packed screen image, not over 32K
             jsl    $E100A8             ; ask GS/OS for the length of the data
             dc     i2'$2019'           ; Get_EOF call
             dc     i4'myCallBlock'

* Now we need a handle to read it into
             pha
             pha                        ; return space
             pea    0                   ; size, high word
             lda    EOFreturned         ; the actual size of the packed data
             sta    actualPackDataSize
             clc
             adc    #65                 ; ask for a handle 65 bytes longer than the data
             pha
             lda    myIDnumber          ; Memory Manager ID for your application
             pha
             pea    $8010               ; attrLocked and attrNoCross
             pea    0
             pea    0                   ; anywhere
             _NewHandle                 ; get the handle
```

Now you have a handle 65 bytes longer than your data that does not cross a bank boundary. You are ready to read in the data and perform a multipass unpack.

PackBytes Buffers Count Too

`PackBytes` can also cause you problems if you do not plan for the worst-case situation. Unlike the other toolbox compression routine `ACECompress`, `PackBytes` is **not** guaranteed to shrink the source data. In fact, your data size may actually grow after a `PackBytes` call.

If you pass a data stream of 64 bytes, all with different values, to `PackBytes`, `PackBytes` puts **65** bytes in your output buffer—the 64 original data bytes and the flag byte of $3F, indicating "64 bytes follow, all different." Unless you preprocess or analyze your data before packing to avoid this situation, make sure your output buffer is large enough to hold the worst case situation, one additional byte generated for every 64 bytes passed to `PackBytes` for compression.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1-3
- File Type Note for File Type $C0, Auxiliary Type $0002, Apple Preferred Format

Apple IIGS
#95: ROM Diagnostic Errors

Written by: Dan Strnad September 1990

This Technical Note describes errors returned by the ROM Diagnostics on Apple IIGS systems.

The Built-In Diagnostics Revealed

The IIGS has a self-test capability in ROM. The self-test is activated by pressing Open-Apple and Option on power up, or Open-Apple, Option, and Reset. During the test, the test number is visible on the bottom of the screen followed by six zeros. After all tests are complete, a continuous 6 KHz one-second beep sounds and the screen displays a `System Good` message. If any test fails, the screen displays a message of the form `System Bad: AABBCCDD` on the lower left hand side and a staggered `AABBCCDD` on the upper left hand side to help read the error code in the event of a RAM failure. In the event of video failure, the failure code is also sent to the printer port. In the number contained in the error message, `AA` is the test number that failed and the failure code is embedded in the `BB`, `CC`, and `DD` fields. The complete failure codes for each of the 12 tests are as follows:

Self Test 1: ROM Test

`AA =`	01
`BB =`	Failed checksum
`DD =`	01 if the test encountered bad RAM and the error code is a RAM error code similar to the RAM Test error codes

For a failure in ROM, the ROM diagnostics also display `RM` on the top left hand corner of the screen.

Self Test 2: RAM Test

`AA =`	02
`BB =`	Bank Number (or $FF for ADB Tool call error)
`CC =`	Bit(s) failed

Self Test 3: Soft Switches and State Register Test

`AA =`	03

`BB =`	State Register bit (if any)
`CC =`	Low byte of soft switch address

Self Test 4: RAM Address Test

AA =	04
BB =	Failed bank number (or $FF for ADB Tool call error)
CCDD =	Failed address

Self Test 5: Speed Test

AA =	05
BB =	01: Speed stuck slow
	02: Speed stuck fast

Self Test 6: Serial Test

AA =	06:
BB =	01: Register R/W
	04: Tx Buffer empty status
	05: Tx Buffer empty failure
	06: All Sent Status fail
	07: Rx Char available
	08: Bad data

Self Test 7: Clock Test

AA =	07
DD =	01: Fatal error occurred and the test is aborted

Self Test 8: Battery RAM Test

AA =	08
BB =	01: Address test and CC = bad address
	02: Non-volatile RAM failed and CC = pattern, DD = address

Self Test 9: Apple Desktop Bus Test

AA =	09
BBCC =	Bad checksum
DD =	01: Apple Desktop Bus tools call encountered a fatal error, no checksum computed.

Self Test 10: Shadow Register Test

AA =	0A
BB =	01: Text page 1 fail
	02: Text page 2 fail
	03: Apple Desktop Bus Tool call error
	04: Power On Clear bit error

Self Test 11: Interrupts Test

`AA =`	0B
`BB =`	01: VBL interrupt time-out
	02: VBL IRQ status fail
	03: 1/4 sec interrupt
	04: 1/4 sec interrupt
	05:
	06: VGC IRQ
	07: Scan line

Self Test 12: Sound Test

`AA =`	0C
`DD =`	01: RAM data error
	02: RAM address error
	03: Data register failed
	04: Control register failed
	05: Oscillator interrupt timeout

Further Reference

- *Apple IIGS Hardware Reference*, Second Edition

Apple IIGS
#96: Standard File Customization

Written by: Dan Strnad November 1990

This Technical Note discusses particulars of using custom dialog boxes for the Open and Save File dialog boxes and custom drawing routines to display the files and folders listed.

About the Templates

Volume 3 of the *Apple IIGS Toolbox Reference* states the following about the Open File dialog box template for Standard File:

> “The scroll bar item (item5) is not used for single-file calls. For multifile calls, this item contains the Accept Button Definition.”

What is not stated explicitly is that, although the scroll bar item is not used for single-file calls, a place holder for it must be included in the dialog box template. Another particular not explicitly stated is that the strings used by the item templates must be Pascal strings; no `listType` field is provided in the extended list control record as was present in the List Manager’s original list record structure.

Custom Item Draw Procedures

Custom item draw procedures have the rectangle in which the item is to be drawn, the List Manager’s `memrec` structure corresponding to that item, and a handle to the extended list control record available on the stack. By including a custom item draw procedure, programs are able to get a handle to the extended list control record. The custom item draw procedure could also make the handle available to other routines, such as the `dialogHook` routine. With the handle, programs can now perform specialized operations during a standard file call, such as checking which item is selected before allowing the user to cancel. The code fragment below (from DTS Apple II Sample Code #18, AccessPriv) illustrates the use of `SFPGetFile2` with a custom item draw routine.

```
static char SaveStr[] = "\pSave";
static char OpenStr[] = "\pOpen";
static char CloseStr[] = "\pClose";
static char DriveStr[] = "\pDrive";
static char CancelStr[] = "\pCancel";
static char FolderStr[] = "\pNew Folder";
static char AcceptStr[] = "\pAccept";
```

```
ItemTemplate OpenBut640 =   {1,
                            61,265,73,375,
                            buttonItem,
                            OpenStr,
                            0,
                            0,
                            0L};

ItemTemplate CloseBut640 =  {2,
                            79,265,91,375,
                            buttonItem,
                            CloseStr,
                            0,
                            0,
                            0L};

ItemTemplate NextBut640 =   {3,
                            25,265,37,375,
                            buttonItem,
                            DriveStr,
                            0,
                            0,
                            0L};

ItemTemplate CancelBut640 = {4,
                            97,265,109,375,
                            buttonItem,
                            CancelStr,
                            0,
                            0,
                            0L};

ItemTemplate Scroll640 =    {5,
                            43,265,55,375,
                            buttonItem,
                            AcceptStr,
                            0,
                            0,
                            0L};

ItemTemplate Path640 =      {6,
                            12,15,24,395,
                            userItem,
                            0L,
                            0,
                            0,
                            0L};

ItemTemplate Files640 =     {7,
                            25,18,107,215,
                            userItem + itemDisable,
                            0L,
                            0,
                            0,
                            0L};

ItemTemplate Prompt640 =    {8,
                            3,15,12,395,
                            statText + itemDisable,
                            0L,
                            0,
                            0,
                            0L};
```

```
/**********************************************************************
*
* myDialogHook
*
***********************************************************************/

pascal void myDialogHook(strip1,strip2)
long strip1;
long strip2;
{
}

/**********************************************************************
*
* CustomItemDraw
*
***********************************************************************/

pascal void CustomItemDraw(itemDrawPtr)
Pointer itemDrawPtr;
{
static unsigned int flag, dbr;        /* result, data bank register value */
byte          StringCount;
char          *ItemPascalString;
Word          ItemFileType;
Long          ItemAuxType;
Rect          *TheItemRectPtr;
MemRec        *TheMemRecPtr;
CtlRecHndl    TheSFListControlHndl;
Point         MyOldPenPos,
              MyNewPenPos;

static char FileString[] = "xxxx yyyyyyyy ";

/* save our data bank and set current to global page */
dbr = SaveDB();
/* Get the Rect from High on the Stack */
TheItemRectPtr = (Rect *)(*((long *)(((long)&itemDrawPtr)+ 36L)));
                                                        /* save old pen position */
GetPen(&MyOldPenPos);                                   /* Set our pen position */
MyNewPenPos.h = TheItemRectPtr->h1 + 5;
MyNewPenPos.v = TheItemRectPtr->v2 -2;
MoveTo(MyNewPenPos);                                    /* relocate the pen */

/* get our member record; this is just to reveal where it is on the stack */
TheMemRecPtr = (MemRec *)(*((long *)(((long)&itemDrawPtr)+ 32L)));

/* get the list cntrol handle; ditto */
TheSFListControlHndl = (CtlRecHndl)(*((long *)(((long)&itemDrawPtr)+ 28L)));

StringCount = (byte) *itemDrawPtr;                      /* get the string length */
ItemPascalString = itemDrawPtr;                         /* set our user string */
ItemFileType =  *(Word *)(itemDrawPtr+StringCount+1L);  /* get our FileType */
ItemAuxType =  *(Long *)(itemDrawPtr+StringCount+3L);   /* get our AuxType */

/* format for display */
sprintf(FileString, "%.4x-%.8lx ",ItemFileType,ItemAuxType);
c2pstr(FileString);                             /* turn it into a P string */
DrawString(FileString);                         /* Draw it */
DrawString(ItemPascalString);                   /* catenate File name to the other
info */
FrameRect(TheItemRectPtr);
MoveTo(MyOldPenPos);                            /* return the pen to starting
position */
RestoreDB(dbr);                                 /* restore our data bank */
```

```
}

/************************************************************************
*
* ChooseFolder
*
* presents user with dialog to select folder to show/set privileges of
*
************************************************************************/

void    SomeProc()
{
DialogTemplate GetDialog640;

GetDialog640.dtBoundsRect.v1 = 0;
GetDialog640.dtBoundsRect.h1 = 0;
GetDialog640.dtBoundsRect.v2 = 114;
GetDialog640.dtBoundsRect.h2 = 400;
GetDialog640.dtVisible = -1;
GetDialog640.dtRefCon = 0L;
GetDialog640.dtItemList[0] = &OpenBut640;
GetDialog640.dtItemList[1] = &CloseBut640;
GetDialog640.dtItemList[2] = &NextBut640;
GetDialog640.dtItemList[3] = &CancelBut640;
GetDialog640.dtItemList[4] = &Scroll640;
GetDialog640.dtItemList[5] = &Path640;
GetDialog640.dtItemList[6] = &Files640;
GetDialog640.dtItemList[7] = &Prompt640;
GetDialog640.dtItemList[8] = 0L;

SFPGetFile2(    /* user selection of folder to get/set privs of */
            120, 53,
            CustomItemDraw,
            refIsPointer,
            prompt,
            0L,
            0L,
            &GetDialog640,
            myDialogHook,
            &myReply
);
```

Further Reference

- *Apple IIGS Toolbox Reference,* Volumes 1 & 3
- DTS Apple II Sample Code #18, AccessPriv

Apple IIGS
#97: Picture Comments and Printing

Written by: Matt Deatherage, Suki Lee & Ben Koning

November 1990

This Technical Note discusses QuickDraw Auxiliary picture comments and how they can be used to help control the printing process.

What's a Picture Comment?

Picture comments are a way in which extra information beyond normal QuickDraw II calls can be embedded in a QuickDraw II picture. Comments can contain virtually anything; they consist of a length, a handle containing the comment and a "kind" that identifies the general type of information in the comment. Picture comment kinds less than or equal to 256 ($100) are reserved for Apple Computer, Inc.

For comments to have any significance, there must be a way that a routine can take special action on them. One of the standard bottleneck procedures is called every time a picture comment is encountered, and it is passed the picture comment's kind, size, and handle on QuickDraw II's direct page. You can insert the address of a custom picture comment handler into the `grafProcs` field of a `grafPort` as described in Apple IIGS Technical Note #34, Low-Level QuickDraw II Routines. If no custom comment handler is present in the `grafPort`, the system calls its own `StdComment` routine, which ignores all comments.

The current picture comment handling routine (either a custom one or the system's default one) is called whenever a picture comment is generated (with the QuickDraw Auxiliary call `PicComment`) or played back from a picture (from within `DrawPicture`). Since the picture comment handling procedure is called when the comment is created, a picture does not have to be open for this facility to work.

Picture comments are ideal ways for applications to pass information to printer drivers as they are generated through toolbox calls and are easily accessible to any desktop program. If the printer driver is printing in immediate mode, it can intercept and act on the picture comment when it is generated. If the printer driver is printing in deferred mode and recording page images with QuickDraw II pictures, it can intercept and act on the picture comment when the picture is played back.

Apple's ImageWriter, ImageWriter LQ and LaserWriter drivers (from System Software 5.0.3) all support various kinds of picture comments for controlling printed output. Applications are encouraged to use these picture comments for finer control over printing. Authors of printer drivers are encouraged to act on these picture comments where appropriate, so applications which use them achieve similar results across printing platforms.

The LaserWriter Driver's Picture Comments

Version 2.2 and later of the LaserWriter driver support the following five PostScript® picture comments:

Name	Kind	Size	Handle
`PostScriptBegin`	190	0	`NIL`
`PostScriptEnd`	191	0	`NIL`
`PostScriptHandle`	192	–	PostScript data
`PostScriptFile`	193	–	PostScript path name
`TextIsPostScript`	194	0	`NIL`

Table 1—PostScript Picture Comments

The print loop must be completed normally with or without any PostScript picture comments that are included. PostScript transmission must begin with the `PostScriptBegin` picture comment and end with the `PostScriptEnd` picture comment. Never nest `PostScriptBegin` and `PostScriptEnd` picture comments.

The `PostScriptHandle` picture comment takes a handle containing PostScript commands (in the form of ASCII data) and sends it to the LaserWriter. The `size` field must contain the size of the handle.

The `PostScriptFile` picture comment takes a handle containing the pathname of a disk file containing PostScript commands. The `size` field must contain the size of the pathname.

The `TextIsPostScript` picture comment takes text drawn through the QuickDraw II `StdText` bottleneck and sends it to the LaserWriter as PostScript. This picture comment has the effect, from the application's point of view, of interpreting all strings passed to `DrawString` and similar calls as PostScript. This picture comment is specific to LaserWriters (`idev = $0003`). Other drivers do not implement this picture comment; therefore, text drawn through QuickDraw II is simply printed—it is neither interpreted as PostScript nor ignored.

The driver does not check for PostScript errors, so the data sent to the LaserWriter must be correct. Always terminate PostScript text with a carriage return character. The transformation the driver uses flips text and prints it upside down on the page. Applications should set their own transformation matrices to serve their needs. Never use the LaserWriter's `userdict`—define a local dictionary for your application's use. Never use `exitserver`, `initgraphics`, `grestoreall`, `erasepage`, or `showpage` PostScript commands, as these commands can alter the driver's environment.

See Chapter 3 of the *LaserWriter Reference Manual* for some examples of how to use picture comments.

The ImageWriter Driver's Picture Comments

ImageWriter driver version 4.0 and later uses three picture comments to control alternate color selection:

Name	Kind	Size	Handle
`Reserved`	250	–	Reserved
`FillColorTable`	251	42	See below
`ChangeSCBs`	252	14	See below

The structure passed in the handle to `FillColorTable` looks like the following:

`version`	word	must be zero
`signature`	word	must be $A55A
`tableno`	word	the color table to be modified (0-15)
`table`	32 bytes	The new color values for the color table
`reserved`	long	must be zero

The structure passed in the handle to `ChangeSCBs` looks like the following:

`version`	word	must be zero
`signature`	word	must be $A55A
`Y1`	word	line number of first SCB to change (from zero to rPage.Y2)
`Y2`	word	line number of last SCB to change (from zero to rPage.Y2)
`SCBvalue`	word	the new scan line control byte
`reserved`	long	must be zero

`PrOpenPage` reinitializes the printing `grafPort`, so these picture comments should be used immediately after `PrOpenPage` to set custom colors. Having lines with both 320- and 640-mode is not recommended and will probably not be supported in the future.

The ImageWriter driver uses picture comment 250 to internally mark the end of a page. Applications must not use this picture comment; use `PrClosePage` to end a page's definition.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1–3
- Apple IIGS Technical Note #34, Low-Level QuickDraw II Routines
- Apple IIGS Technical Note #35, Printer Driver Specifications
- Apple IIGS Technical Note #93, Compatible Printing
- *d e v e l o p*, October 1990, Issue 4, "Driving to Print"

PostScript is a registered trademark of Adobe Systems, Incorporated.

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#98: Aren't Windows A Pane?

Revised by: Dave Lyons May 1992
Written by: Dave Lyons January 1991

This Technical Note describes interesting Window Manager things.
Changes since January 1991: Noted that in System 6.0 it's safe to use Window color table resources. Added a section on changing the desktop pattern or picture.

Changing the Desktop Pattern or Picture

The best way to set a new desktop pattern or picture is as follows. This works with the Finder and other desktop applications.

1. Use `MessageCenter` to delete message 2, the desktop message. (If there wasn't one, that's fine—there still isn't.)
2. Use `MessageCenter` to create a new message 2, containing the pattern or picture you want (see the Window Manager chapter of *Apple IIGS Toolbox Reference*, Volume 3).
3. Call `Desktop` (in the Window Manager) with a `deskTopOp` of 8 and a `dtParam` of $00000000. This notifies any part of the system that cares (such as the Finder) that there is a new desktop pattern.
4. Call `Desktop` with a `deskTopOp` of 4 and a `dtParam` of $00000000 and keep the result.
5. Call `Desktop` with a `deskTopOp` of 5 and use the result from step 4 as `dtParam`. This sets the desktop pattern to what it already is, forcing the desktop to redraw (this works whether a pattern, picture, or pointer to desktop-drawing routine is involved).

A Warning About Window Color Table Handles And Resources

The System 6.0 Window Manager fixes the problem described below. If your application requires System 6, you can safely ignore this section.

All versions of the Window Manager that support window color tables specified as handles or resources, up to and including System Software 5.0.4, work unreliably when a standard window's color table is supplied by handle or resource ID.

The problem is not immediately obvious; only one bit of memory is accidentally cleared, but the address is unpredictable in advance. (When unlocking the color table handle, the standard window definition procedure attempts to unlock the handle manually by turning off bit 15 of word offset +4 in the master pointer record. But it gets the high and low words of the handle reversed and usually turns off bit 15 of the word at offset $80E4 or $80E5 in some bank of RAM determined by the low byte of the handle.)

The solution is to avoid supplying color table handles or resource IDs to the Window Manager. Supply color table pointers instead. You can get a color table pointer from a color table resource ID by calling `LoadResource` on the color table resource, locking the handle and dereferencing it. Memory is less fragmented if color table resources used in this way are marked as `attrFixed`.

One method is to put the window color table pointer into the window template before calling `NewWindow2`. If you are creating the window from an `rWindParam1` resource, you need to use `LoadResource` to get the template into RAM so that you can stuff the color table pointer into the template. (Be sure to change the `moreFlags` field to indicate that the color table is a pointer, if the template indicates it's a resource.) After you create the window with `NewWindow2` (by handle), use `ReleaseResource` to release the `rWindParam1` resource.

Another method is to create the window as invisible and pass the window color table pointer to `SetFrameColor` before calling `ShowWindow`.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 2-3

Developer Technical Support

Apple IIGS
#99: Supplemental Scrap Types

Revised by: Dave Lyons May 1992
Written by: Matt Deatherage & Matthew Reimer January 1991

This Technical Note describes public scrap types.
Changes since March 1991: Added information on Scrap Type $8003 (Resource Reference Scrap); added a cross-reference to HyperCard IIGS Technical Note #3.

The *Apple IIGS Toolbox Reference* lists only two known scrap types—text ($0000) and pictures ($0001). Other assigned scrap types are documented in this Note. The format used to describe the scraps 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 scrap handle.

Sampled Sound Scrap (Type: $0002)

The following describes the Sampled Sound scrap format. It consists of a ten-byte header followed by the sample data bytes. This format is identical to the sampled sound resource format.

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, Tuning Sampled Sounds.)
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 scrap.

TextEdit Style Scrap (Type: $0064)

The TextEdit Style Scrap format is the same as the `TEFormat` structure defined in Volume 3 of the *Apple IIGS Toolbox Reference*, which is also the same as the `rStyleBlock` resource format defined in the same volume.

Icon Scrap (Type: $4945)

The Icon scrap format is the same as the format for Finder Icon Data records, documented in detail in the File Type Note for File Type $CA, Finder Icon Files. If there is more than one Icon Data record in a scrap, they are concatenated together with no intervening space.

Mask Scrap (Type: $8001)

The Mask scrap format is exactly the same as the PICT scrap ($0001) format, except that the pixel image the picture draws contains only zeroes and ones. When drawn, this picture creates a mask. The mask has zeroes where the image can be seen through the mask, and ones where the mask does not allow the picture through. When pasting a Mask scrap, initialize the destination bitmap to zero and draw the picture.

You can create the mask image by using regular QuickDraw II calls (using ovals, rectangles, etc.) or you can create it independently and include it with `PaintPixels` or other pixel map manipulation routines.

Color Table Scrap (Type: $8002)

The following describes the Color Table scrap format. The scrap contains color tables so that applications can keep custom colors with pictures copied to the clipboard. The scrap has the same format as the Apple Preferred Format picture `PALETTES` block:

NumColorTables	(+000)	**Word**	The count of the number of color tables in the scrap
ColorTableArray	(+002)	**32 Bytes**	The color tables for the scrap. There are NumColorTables of them, each 32 bytes long.

Resource Reference Scrap (Type: $8003)

The Resource Reference scrap is designed to allow resource editors to exchange resource data through an external scrap file using the Scrap Manager.

resScrapType	(+000)	**Word**	Type of resource (within the resScrapPath file)
resScrapID	(+002)	**Long**	ID of resource (within the resScrapPath file)
resScrapPath	(+006)	**WString**	Full GS/OS class-one pathname to an extended file containing the specified resource.

If the specified resource contains references to other resources (for example, an `rWindParam1` resource with a title string, control list, control templates, etc.), all the referenced resources must be present in the `resScrapPath` file.

It is the responsibility of the application using this scrap to handle resource ID conflicts that might arise from a Paste operation. The application should not modify or destroy the `resScrapPath` file.

Further Reference

- *Apple IIGS Toolbox Reference*
- HyperCard IIGS Technical Note #3, Tuning Sampled Sounds
- File Type Note for file type $CA, all auxiliary types, Finder Icons File
- File Type Note for file type $C0, auxiliary type $0002, Apple Preferred Format

Apple II
Technical Notes

Developer Technical Support

Apple IIGS #100: VersionVille

Revised by: Matt Deatherage — May 1992
Written by: Matt Deatherage — January 1991

This Technical Note is all there is to know about versions, version formats and version numbers on the Apple IIGS.
|**Changes since January 1991:** Revised to include System Software 6.0.

Version Number Formats

There are three kinds of version numbers on the Apple IIGS. Two of the three are documented elsewhere but are repeated here for convenience.

System Tool Set Versions

The Apple IIGS system tools use a one-word version number. The high-order **four** bits of this word have special meaning. Bits 8-11 are the major version number and bits 0-7 are the minor version number. This is illustrated in Figure 1.

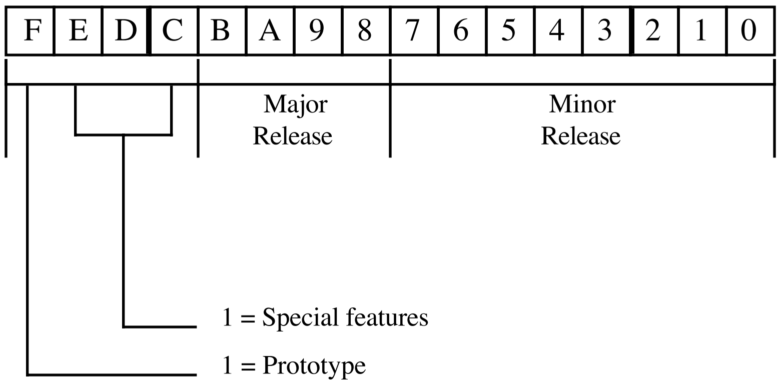

Figure 1—Toolbox Version Numbers

Note that this definition is different and supersedes the definition in the *Apple IIGS Toolbox Reference* for system tool sets. Previous documentation reserves only bit 15 as the prototype bit; this has been expanded. Bits 14-12 of user tool set version words have no special meaning; they are still part of the major release.

Note: When comparing the major and minor release version numbers to check the installed version of a system tool, mask off bits 15-12 first (for example, by using an `AND #$0FFF` instruction).

SmartPort Or GS/OS Driver Versions

GS/OS drivers and SmartPort firmware drivers use an alternate one-word version number, described in Figure 2.

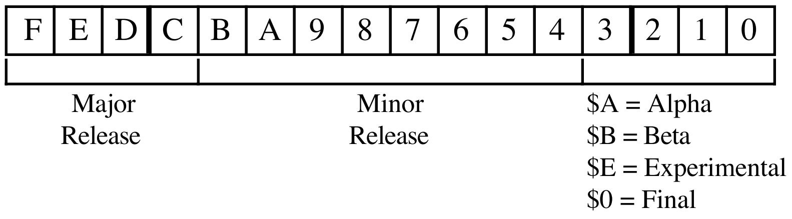

Figure 2—GS/OS Driver And SmartPort Version Numbers

Apple IIGS Long Version Format

Long version format is a 32-bit (two-word) format similar to the standard Macintosh version numbering scheme defined in Macintosh Technical Note #189, Version Territory, except the four bytes are stored least significant byte first, as is standard on the Apple II, and the values of the stage are different. Figure 3 shows the format of a long version.

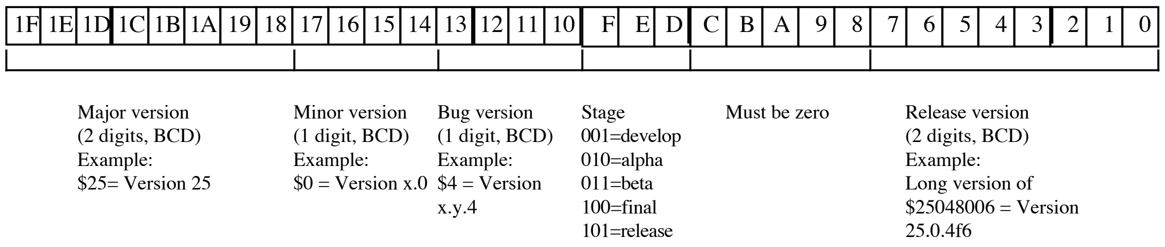
1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F E D C B A 9 8 7 6 5 4 3 2 1 0
Major version (2 digits, BCD) Example: $25= Version 25
Minor version (1 digit, BCD) Example: $0 = Version x.0
Bug version (1 digit, BCD) Example: $4 = Version x.y.4
Stage 001=develop 010=alpha 011=beta 100=final 101=release
Must be zero
Release version (2 digits, BCD) Example: Long version of $25048006 = Version 25.0.4f6

Figure 3—Long Version Numbers

Long version format allows for bug versions, unlike toolbox versions. Also, you can do unsigned long comparisons of long versions to determine which revision is later.

Note: If the version stage is 101 (release), the release version **must** be zero. For example, you may not have version 25.0.4 release 16. "Release version" implies that the product is no longer under development and has no developmental version numbers.

System Version Numbers

The most important of the numerous version numbers in the system are the system tool version numbers. These numbers, passed to `LoadTools`, `LoadOneTool` or `StartUpTools` ensure that you're getting at least the version you want, or maybe a later one. This mechanism is your primary defense against old system software—by requiring the latest tool versions in your application, you are notified by the Tool Locator early in your program if the system has the latest system software installed or not.

Note that ROM 1 and ROM 3 have different version numbers for seven tools under 5.0.4—QuickDraw II, the Scheduler, ADB, SANE, Integer Math, Text Tools and the List Manager. In each case, the ROM 01 version is lower and should be used in your `LoadOneTool`, `LoadTools` or `StartUpTools` calls.

The current revision of Apple IIGS System Software is 6.0. Assuming a correct installation, requiring QuickDraw 3.7 in effect requires System Software 6.0, although you may check the system's `rVersion` resource in the system resource file if you require more detailed information about the system sovtware version.

System Tool Set Versions

Number	Tool	ROM 1	ROM 3
1	Tool Locator	$0301	$0301
2	Memory Manager	$0302	$0302
3	Misc Tools	$0302	$0302
4	QuickDraw II	$0307	$0307
5	Desk Manager	$0304	$0304
6	Event Manager	$0301	$0301
7	Scheduler	$0300	$0300
8	Sound Tools	$0303	$0303
9	ADB	$0300	$0300
10	SANE	$0300	$0300
11	Integer Math	$0300	$0300
12	Text Tools	$0300	$0300
13	[used internally]	$0300	$0300
14	Window Manager	$0303	$0303
15	Menu Manager	$0303	$0303
16	Control Manager	$0303	$0303
17	[System Loader]	$0400	$0400
18	QuickDraw II Aux	$0304	$0304
19	Print Manager	$0301	$0301
20	Line Edit	$0303	$0303
21	Dialog Manager	$0304	$0304
22	Scrap Manager	$0301	$0301
23	Standard File	$0303	$0303
25	Note Synthesizer	$0104	$0104
26	Note Sequencer	$0104	$0104
27	Font Manager	$0303	$0303
28	List Manager	$0303	$0303
29	ACE	$0103	$0103
30	Resource Manager	$0102	$0102
32	MIDI Tools	$0103	$0103
33	Video Overlay	$0103	$0103
34	Text Edit	$0103	$0103
35	MIDI Synth	$0100	$0100
38	Media Control	$0100	$0100

Toolbox Driver Version Numbers

Driver	Version
ImageWriter II	4.2
ImageWriter LQ	4.2
LaserWriter	3.2
StyleWriter	1.0
Epson	2.0
Printer Port Driver	2.1
Modem Port Driver	2.1
Parallel Card Port Driver	2.0
AppleTalk Port Driver	3.0
Pioneer 4200 (MC)	1.0
Pioneer 2000 (MC)	1.0
Apple CD SC (MC)	1.0

GS/OS Version Numbers

Component	Version
GS/OS	4.1
ProDOS FST	4.1
AppleShare FST	4.0
High Sierra FST	4.0
Character FST	4.0
DOS 3.3 FST	1.2
HFS FST	1.0
Pascal FST	1.0
AFP Driver	4.0
Apple II RAMCard driver	1.0
AppleDisk 3.5 Driver	5.3
AppleDisk 5.25 Driver	2.5
AppleTalk Main Driver	4.0
Console Driver	3.2
RPM Driver	4.0
SCSI CD Driver	6.0
SCSI HD Driver	6.0
SCSI Scanner Driver	6.0
SCSI Tape Driver	6.0
UniDisk 3.5 Driver	3.0

Control Panel Version Numbers

CDev	Version
AppleShare	2.0
Direct Connect Printer	1.1
FolderPriv	1.0
General	2.0
Keyboard	1.1
Media Control	1.1
MIDI	1.0
Modem Port	1.1
Monitor	1.1
Network Printer Namer	1.0
Network Printer Chooser	1.0
Network	1.0
Printer Port	1.1
RAM	1.1
SetStart	1.0
Slots	1.2
Sound	2.0
Time	2.0

Further Reference

- *Apple IIGS Toolbox Reference*
- *GS/OS Reference*
- GS/OS Technical Note #1, Contents of System Disk and System Tools
- File Type Note for File Type $C7, Control Panel Devices

Apple II
Technical Notes

Developer Technical Support

Apple IIGS #101: Patching the Toolbox

Revised by: Eric Shepherd December 2000
Written by: Dave Lyons May 1991

This Technical Note presents guidelines on when and how to patch Apple IIGS Toolbox functions.
Changes since May 1992: Noted a system bug preventing the automatic removal of Toolbox function patches from working correctly.

Introduction

There is normally no need to patch the toolbox; avoid patching whenever you can. If you must patch a toolbox function, be sure to have a good understanding of the call you're patching and how it interacts with the whole system.

No toolbox patch is risk-free. Future versions of the toolbox could change in ways that make your patch less useful. (For example, if you patched `NewControl` to have some global effect on controls being created, your patch became less useful when `NewControl2` was introduced in System Software 5.0.)

For better compatibility, patch with care! If any parameters passed are outside the range that was allowed when you wrote your patch, just pass the call straight through; the new toolbox probably knows something your patch doesn't.

Patching the Toolbox From an Application

An application can easily patch a function for the duration of that application.

After starting up the tools, construct a Function Pointer Table (FPT) the same size as the existing FPT (call `GetTSPtr` and examine the first word of the table; multiply it by four to get the size of the FPT in bytes). The first longword of your FPT is the number of functions in the tool set; do not hard-code this value! Get it from the existing FPT on the fly. Fill the rest of your FPT with zeroes, except for the functions you want to patch. You must always patch the BootInit function (the first function) to return no error. Remember that the function pointer values are one less than the addresses of your replacement functions.

On exit, when you call `TLShutDown` your patch will be automatically removed. (If you're using `ShutDownTools`, you should call `MMShutDown` and `TLShutDown` after you call `ShutDownTools`.)

Note: Toolbox function patches are not actually correctly removed from the system at `TLShutDown` time. Instead, you will have to save the original FPT when before your application patches the Toolbox functions, then restore it before your application exits.

Note: In the description of `SetTSPtr` on page 24-19 of Apple IIGS Toolbox Reference, Volume 2, there are several references to the TPT. Keep in mind that the TPT is the Toolset Pointer Table, not the Function Table Pointer you pass to `SetTSPtr`. While `SetTSPtr` copies the TPT to RAM if necessary, it does not make a copy of the FPT. After you call `SetTSPtr`, the FPT you passed is being used, and any zero values in your table were filled in.

Patching the Toolbox From a Desk Accessory or Setup File

A permanent initialization file or Desk Accessory can patch toolbox functions at boot time by constructing an FPT for `SetTSPtr`, as described for an application, but there is an extra step to make the patch "stick."

Call `LoadOneTool` and then `SetTSPtr`; then call `SetDefaultTPT` (see *Apple IIGS Toolbox Reference* Volume 3, page 51-16).

It is not safe to call `SetDefaultTPT` while an application is running (temporary application patches would be made permanent, and later the application would go away). Since there are desk accessories that install other desk accessories while applications are running, desk accessory that wants to install a tool patch should make the class-one GS/OS `GetName` call; if the null string is returned, no application is executing yet, so it is safe to make the patch. (Otherwise the desk accessory should ask the user to put the desk accessory file in the System:Desk.Accs folder and restart the system.)

Patching the Tool Locator or Desk Manager

On ROM 3 systems, the `SetTSPtr` call treats toolsets 1 (Tool Locator) and 5 (Desk Manager) specially, for compatibility with system software versions earlier than 5.0.

You must pass a `systemOrUser` value of $0001 (not $0000) when patching one of these toolsets, or the `SetTSPtr` call will have no effect. Passing this special `systemOrUser` value works for other ROM versions, too—you don't have to check the ROM version.

Avoid Tail Patching

The best kind of patch is a pre-patch or head patch: it does some extra work and then jumps to the original function (as found in the FPT before applying the patch). Make sure the A, X, and Y registers contain the same values when you jump to the original function as they did when the patch got control.

A "tail patch" which calls the original function and then regains control is much more of a compatibility risk, because there are several instances where System Software patches examine return addresses to fix problems in large toolbox calls which call small ones (by patching the small one to realize it's being called from the big one, many K of RAM remain available to your application).

If you tail patch a function which the system already patched, you may prevent the toolbox from working correctly.

Patching the Tool Dispatcher

If you need to patch a large number of functions, especially for a general purpose utility like a debugger, it may make more sense to patch the tool dispatcher vectors instead of patching individual functions. See Apple IIGS Technical Note #87, Patching the Tool Dispatcher.

Further Reference

- *Apple IIGS Toolbox Reference*
- Apple IIGS Technical Note #87, Patching the Tool Dispatcher

Developer Technical Support

Apple IIGS
#102: Various Vectors

Revised by: Dave Lyons May 1992
Written by: Dave Lyons December 1991

This Technical Note describes system vectors that are not fully described in other documentation.
Changes since December 1991: Added information about the TOBRAMSETUP vector.

The `TOBRAMSETUP` vector

The `TOBRAMSETUP` vector is documented in Appendix D of the *Apple IIGS Firmware Reference*. Two clarifications are needed:

- `TOBRAMSETUP` must be called in 8-bit native mode (SEP #$30).
- Before System 6.0, `TOBRAMSETUP` required that the Bank register be $00 (bad things would happen if it was not). This requirement is gone in 6.0.

The MOVE_INFO vector

`MOVE_INFO` is a flexible, low-overhead data transfer routine. It can transfer buffer-to-buffer, buffer-to-location, location-to-buffer, and buffer-to-buffer reversing the order of the bytes.

Apple IIGS GS/OS Device Driver Reference tells you how to call `MOVE_INFO` from a GS/OS driver environment (`JSL` to $01FC70), but this requires the language-card RAM to be banked in correctly.

Another vector points to the same routine: $E10200. If you aren't a GS/OS device driver, it is more convenient to `JSL` to $E10200, because you don't have to worry about banking in the $01FCxx vectors. The $E10200 vector is available whenever GS/OS is active, under System Software 5.0 or later.

The DYN_SLOT_ARBITER and SET_SYS_SPEED vectors

Two other GS/OS System Service vectors are duplicated in bank $E1: `SET_SYS_SPEED` ($E10204) and `DYN_SLOT_ARBITER` ($E10208). Like `MOVE_INFO`, these are available when GS/OS is active under System Software 5.0 or later.

Further Reference

- *Apple IIGS GS/OS Device Driver Reference*
- *Apple IIGS Firmware Reference*

Apple IIGS
#103: Inline Procedure Name Format

Modified by: Matt Deatherage — May 1992
Written by: Dave Lyons — December 1991

This Technical Note describes a simple format for imbedding procedure names in object code, for use by debugging utilities.
Changes since December 1991: Changed `&syscnt` to `&SYSCNT` so it works with the `CASE ON` APW directive. Clarified the possible addition of parameters after the Pascal string.

GSBug 1.5b18 and later support a simple convention for including procedure names inline in the object code, for debugging purposes.

Inline Name Format

```
82 xx xx                          brl pastName
71 77                             dc.w $7771
nn xx xx xx xx...                 str 'the name string'
                      pastName    ...
```

That is, an imbedded name is a `BRL` around a signature word and a Pascal string. The name string can theoretically be up to 255 characters long, but in practice only short names are useful. For example, GSBug displays only the first 15 characters of a name when it is encountered, and only the first 11 when it appears as the operand of a `JSR` or `JSL` instruction.

Names in this format always start with a `BRL`, not a `BRA` or `JMP`. Signature word values other than $7771 are **reserved** for future definition, and more information may be added after the Pascal string.

Be careful what you name!

Be careful not to name something important—like a table, or a label from which you compute other addresses. The extra bytes generated by the inline name would mess up your calculations. If you name a heartbeat task, out-of-memory queue routine, or other construction that needs a special header, be sure to put the name where the executable code starts, not at the beginning of the header.

APW Assembly Macro

The following macro is for the APW assembler. If you equate `DebugSymbols` to zero, the macro generates no object code. If `DebugSymbols` is nonzero, the macro generates an inline name corresponding to its label.

Use the `name` macro anywhere you would use a label. For example:

```
DebugSymbols   GEQU 1
...
CountItems     name
```

The macro:

```
     MACRO
&lab name
&lab anop
     aif DebugSymbols=0,.pastName
     brl pastName&SYSCNT
     dc i'$7771'
     dc i1'L:&lab',c'&lab'
pastName&SYSCNT anop
.pastName
     MEND
```

MPW IIGS Assembly Macros

The following macros are for the MPW IIGS assembler. If you equate `DebugSymbols` to zero, the macros generate no object code. If `DebugSymbols` is nonzero, the macros generate inline names corresponding to their labels.

Use the `name` macro anywhere you would use a label. Use the `procname` macro in place of a `proc` directive, at the beginning of a procedure. For example:

```
DebugSymbols      equ 1
...
CountItems        name
TaskLoop          procname
```

The macros:

```
           macro
&lab       name
&lab
           if DebugSymbols<>0 then
           brl @pastName
           lclc &olds
&olds      setc &setting('string')
           string asis
           dc.w $7771
           dc.b &len(&lab),'&lab'
           string &olds
@pastName
           endif
           mend

* You can use procname instead of proc

           macro
&lab       procname   &x
&lab       proc       &x
           if DebugSymbols<>0 then
           brl @pastName
           lclc &olds
&olds      setc &setting('string')
           string asis
           dc.w $7771
```

```
                 dc.b &len(&lab),'&lab'
                 string &olds
      @pastName
                 endif
                 mend
```

Writing utilities that recognize inline names

If you write a utility that recognizes inline procedure names in this format, check for a signature word of $777x, not specifically $7771. This allows more information to be added to the format later (a signature of $7772 could mean there is a Pascal string followed by parameter-passing information, for example).

Apple IIGS
#104: Font Manager Fundamentals

Written by: Matt Deatherage December 1991

This Technical Note discusses information and philosophy of that typographical toolset, the Font Manager.

FixFontMenu only works once per FMStartUp

You may have noticed that none of the Font Manager calls that translate font family numbers to menu item IDs (or vice-versa) require a menu ID as a parameter. That's because the Font Manager was designed with the idea that an application would only need one font menu, so it keeps one correspondence in private static storage.

This means that once someone has called `FixFontMenu`, any later `FixFontMenu` call during that Font Manager session will destroy the results of the first one, unless all the parameters are identical. The Font Manager doesn't remove the font menu items, but it does not return the correct results from `FamNum2ItemID` or `ItemID2FamNum`.

This means if you're a new desk accessory, the Font Manager can't help you create a font menu—attempting to use `FixFontMenu` will make any application font menu useless. You can use Font Manager routines such as `CountFamiles`, `FindFamily` and `GetFamInfo` to obtain all the information necessary to build your own font menu (or font choosing dialog box, for that matter—but if you create a dialog for an NDA, remember that it has to fit in 320 mode also).

Font styling requires QuickDraw Auxiliary

The Font Manager can't create fonts with outline, shadow or italic styles unless QuickDraw Auxiliary (tool set #18) is present and started. These facts are mentioned in pieces other places, but not in one place—if you want normal Font Manager operations, you must load and start QuickDraw Auxiliary.

Further Reference

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

Apple IIGS
#105: We Interrupt This CPU...

Written by: Matt Deatherage May 1992

This Technical Note supplements the discussion of how interrupts generally work (or don't work) on the Apple IIGS found in the *Apple IIGS Firmware Reference*. It also discusses how to patch into the interrupt chain and when not to use software interrupts.

This Note is a Supplement

That's right, a supplement. This is not the definitive, end-all discussion of interrupts on the Apple IIGS. Most of the information you need to know is available, and has been for several years, in the *Apple IIGS Firmware Reference*. If you're going to write an interrupt routine, you **need** to read Chapter 6 of the *Firmware Reference.*

No excuses. If you don't have the book, buy it or borrow it. People who use your software don't want to hear a sad story about how you wanted to spend the money on a couple of CDs instead of preventing their machine from crashing.

If you haven't read Chapter 6 of the *Firmware Reference*, do so before continuing; the rest of this Note will make much more sense if you're familiar with the material covered in that chapter.

A Note About Timing

There are lots of times listed in this Note, concerning how fast certain kinds of interrupts must be serviced before they're lost. Please remember that all times listed are **ideal** times—actual times are likely to be shorter. For example, a maximum response time of a millisecond means you have one millisecond from the time the peripheral asserts the `/IRQ` line until the interrupt must be serviced. If interrupts are disabled for the first 750 microseconds (μs) of that, then your maximum response time is 250 μs. This is why we constantly remind programmers to keep interrupts disabled for absolutely the shortest time possible. Also, all times reflecting serial or AppleTalk interrupts already take into account the serial chip's internal 3-byte buffer.

So What the Heck Are All Those Vectors?

At first, looking at all those various vectors seems pretty darned intimidating. However, the structure becomes clearer when you think about interrupt priority.

Some microprocessors allow interrupt requests to have priorities—higher priority interrupts can interrupt lower priority ones. The 65816 doesn't have this capability, so the best the Apple IIGS can do is check possible interrupt sources in highest-priority-first order. For example, AppleTalk interrupts must always be processed extremely quickly—from the time an AppleTalk interrupt is asserted, someone must read the data from the SCC within a maximum of 104.167 μs or data can be lost. That's not very much time at all, especially considering that the system may have interrupts disabled, or may be running at 1 MHz speed when the interrupt fires.

Serial interrupts are next—at 19,200 baud, there's a maximum of 1.094 milliseconds to read data before it's lost. (Multiplication shows that 38,400 baud has a maximum of 547 μs, and 57,600 baud has a maximum delay of 273.5 μs. Not much at all.)

You'd hope the Interrupt Manager in ROM would be smart enough to service AppleTalk interrupts first and serial interrupts next, and in fact that's what it does. In fact, it services them so fast that not all the system information is saved before checking the hardware and dispatching (if necessary) to the `IRQ.APTALK` or `IRQ.SERIAL` vectors. See Apple IIGS Technical Note #24 for more information on which system state information isn't saved before calling those vectors.

The list of interrupt priorities is on page 180 of the *Firmware Reference*. What's not clear from any description of interrupt handling is that each internal interrupt source's vector is only called if the Interrupt Manager determines it is the source of the interrupt. For example, the `IRQ.DSKACC` vector is **not** called unless the user pressed Command-Control-Esc to generate the interrupt. This insures that external interrupt handlers for slot-based peripherals are dispatched to as quickly as possible—if each vectored routine had to determine interrupt ownership, every interrupt would have significantly more overhead.

There are two additions to the priority list in the *Firmware Reference*—the first is also an exception to the "interrupt handlers don't have to identify the interrupt" rule. On ROM 3 machines only, vector $E1021C (`IRQ.MIDI`) gets control immediately after determining the interrupt isn't an AppleTalk interrupt. MIDI data can come in so quickly that it needs higher priority than serial interrupts. However, to improve performance, routines called through this vector must return as fast as possible (faster would be better) to avoid delaying interrupts further down the chain, like serial interrupts. Also note that this vector doesn't exist on ROM 1.

The second addition is to the final priority, simply defined as "external slot." The documentation doesn't clearly indicate how this works—it kind of implies this is just calling `IRQ.OTHER`. In fact, if no `IRQ.OTHER` routine claims the interrupt, the system does some voodoo magic to switch to emulation mode and jumps through the vector at $03FE, just like all previous Apple II models. And just like in older systems, whatever code is pointed to by $03FE must end with an `RTI` instruction. This behavior is preserved for compatibility, although it is the slowest interrupt response available on the IIGS.

Getting Control In Time

Passing control to external handlers isn't always quick enough for some people. If you're writing a telecommunications program, for example, you have no more than 1.094 ms from the time a character is received to get it out of the SCC or you'll lose data at 19,200 baud.

The Interrupt Manager is a very tight piece of code—if it were running in RAM and the system was temporarily slowed down to 1 MHz, there would only be room for about two more instructions before AppleTalk would lose data. Since AppleTalk has to be serviced within 104.2 µs (as discussed previously), and since `IRQ.SERIAL` is called as quickly as possible after `IRQ.APTALK` (the only delay is if you're on ROM 3 and a non-trivial MIDI interrupt handler is installed), patching in at `IRQ.SERIAL` poses no problems for most high-speed communications, even up to 57,600 baud. In other words, it's not necessary to patch any vector other than `IRQ.SERIAL` to achieve the results you want.

The problem comes when you have external communications hardware—making it through the internal interrupt chain is too slow if your external communications hardware has the same kinds of limitation the SCC does (namely, a 3-byte internal buffer). External vectors are only called after all the internal sources verify it's not their interrupt, and by that time your card may have lost data.

Patching the Main Interrupt Vector

In these cases, where there is no possible way to service an interrupt in time through the Interrupt Manager's normal priority chain, and in these cases **only**, it's acceptable to patch out the main interrupt vector at $E10010 (preferably using `GetVector` and `SetVector` with reference number $0004). But even then, there are rules to follow.

1. You should duplicate the functionality of the main interrupt vector exactly until the point where you **must** gain control or lose data. For example, if your card requires that you service interrupts within a millisecond or lose data, AppleTalk interrupts still have higher priority over your interrupts because AppleTalk interrupts must be serviced within 104 μs. In this example case, your code should duplicate the functionality of the Interrupt Manager up through and including the call to `IRQ.APTALK`, and then (and **only** then) call your interrupt handler, where you handle the interrupt if it's yours and pass control to the rest of the interrupt chain if it's not.

2. You should only service your interrupts before AppleTalk if your interrupts require servicing in less than 104 μs. If they don't, give AppleTalk first shot. If they do, you must **clearly** inform the user, both in documentation and on the screen, that if they proceed with this function network services may be interrupted, and that they may have to restart the system to restore them. Users must also have the option to back out and cancel at this point. No, this isn't a pleasant message to deliver, but it's much nicer than to completely disconnect AppleTalk and lock up the system if it was booted from a server.

3. You should only patch out the main interrupt vector when absolutely necessary. For example, if you're communicating with hardware that runs at multiple speeds and only the highest speed generates interrupts that require patching the main vector, you should **not** be patching the main vector when not using that highest speed. For telecommunication programs, this means different interrupt handling routines depending on baud rates. To do this any other way lessens the reliability of other high-speed interrupt-driven peripherals in the system.

And remember, it's only acceptable to patch the main interrupt vector when there is no other way to service interrupts fast enough. At all other times, even in the same program, service your interrupts in other ways.

Vectors vs. Binding vs. Allocating

There are three main ways to get into the IIGS interrupt-handling chain—by patching vectors directly, by using the ProDOS 8 or ProDOS 16 call `ALLOC_INTERRUPT`, and by using the GS/OS call `BindInt`. Each behaves differently and has advantages and disadvantages. We'll go from the highest level to the lowest in discussing them.

`BindInt`—easy to use, but not as easy to control

`BindInt`'s vector reference numbers (VRNs) are designed to correspond to vectors in the IIGS Interrupt Manager's chain. Comparing the list of numbers on page 265 of *GS/OS Reference* to the list of vectors starting on page 266 of the *Apple IIGS Firmware Reference* will make this more obvious.

When you call `BindInt`, GS/OS replaces the address in the appropriate interrupt vector with an address inside GS/OS. The routine it points to calls all the routines bound to that vector, including the one that was originally installed (usually the ROM's built-in `SEC/RTL` address). That is, if `IRQ.VBL` pointed to the Miscellaneous Tools' Heartbeat Task code before a program made four separate `BindInt` calls to VRN $000C, then after those calls completed, `IRQ.VBL` would point to code inside GS/OS that called all four bound routines and the Miscellaneous Tools' Heartbeat Task code.

This is why each bound routine is told (through the microprocessor's carry flag) if one of the other routines has already claimed the interrupt and why preserving that status is important. `BindInt` is a convenient way to get code time during various kinds of interrupts, but you should note that you can't control in what order bound handlers are called.

`ALLOC_INTERRUPT`—old style interrupt management

`ALLOC_INTERRUPT` and the ProDOS 8 equivalent, `ALLOC_INT` take the address of the routine you pass and keep it in an internal table. When an interrupt occurs, each address in the table is called in turn until one of the interrupt handlers claims it. In older days, failure by any of the installed interrupt handlers to claim the interrupt would bring the system to a crashing halt—nowadays unclaimed interrupts are ignored by both ProDOS 8 and GS/OS.

What the manuals **don't** tell you is that any routine installed in this way is called after the system has jumped through address $03FE in bank zero—in other words, at the last possible chance. For any kind of timing-sensitive interrupts, these routines are not sufficient.

The table that stores these routines is of a fixed size—ProDOS 8's table holds four routines, and GS/OS's holds 16. If you try to install more handlers than that, you'll get an error from the operating system.

Patching Vectors—high level of control, high risk

The lowest level at which you can get control is by directly patching the Interrupt Manager's vectors as documented in the *Firmware Reference*. Although this lets you get control as soon as the Interrupt Manager determines which vector to call, it also carries some compatibility risks.

Any `BindInt` calls with VRNs that reference a vector you patch make GS/OS take your routine's address and store it internally. This is a problem for anyone who daisy-chained into the same interrupt vector after you did—there's no good way to disconnect yourself without disconnecting everyone who patched in after you. This is Bad.

If you patch vectors directly, you have to check the vector when you're ready to remove your routine. If the vector doesn't still point to your address, someone else has patched into the vector after you and you can't remove yourself. In these cases, you have to leave a "code stub" that takes no action other than passing control along to the address that was installed when you patched in, and you have to leave that code stub at the same address as your interrupt handler. (Since you don't know who has patched the vector after you, you have no way to communicate with those programs and tell them you're going away.)

This means your interrupt handler can't be in your main program. If it is, when GS/OS calls `UserShutDown` to remove your program from memory, you'll orphan one or more pointers to your interrupt handler (which doesn't exist anymore). You must allocate memory and load your interrupt handler with a different user ID than your main program so your code stub can survive when your program quits. Also note that this means repeated launchings of your program could leave lots and lots of code stubs in memory—so if you can find a way other than patching vectors directly, you're encouraged to use it.

Software Interrupts—BRK and COP

Sometimes developers forget that `BRK` and `COP` instructions are in fact software interrupts—when the IIGS's 65816 encounters one of these instructions, it goes through the same Interrupt Manager procedures that all interrupts go through.

Among other things, this means that encountering one of these instructions inside an interrupt routine will overwrite all the system's saved information (such as registers or system state variables) with new ones, meaning you'll never be able to return from the first interrupt. This isn't too much of a problem with `BRK` (except when debugging interrupt routines), but a recent fad popularity for `COP` makes this worth mentioning.

Some developers are trying to use `COP` instructions for all kinds of general-purpose mechanisms, but the system is not designed to handle this. Using a `COP` instruction to pass control to a shell or a library routine in production-level code is not acceptable for several reasons. First, any `COP` instruction inside an interrupt handler will bring the system to its knees. Second, there is no arbitration for the `COP` vector so multiple users of it will collide. Third, although a `COP` instruction takes only two bytes, it takes many **hundreds** more cycles to execute than a `JSL` instruction, slowing the system down for no reason.

`COP` instructions are perfectly acceptable in non-production level (debugging) code, but developers should not use them as a way for different program modules to communicate. Such use is not supported and is strongly discouraged by Apple.

Before we RTI—A Summary

This Note covers many issues concerning interrupts, so here's a summary. This isn't all the explanation—refer to individual topics for discussions and reasons.

- Never disable interrupts for longer than necessary—you make life really difficult on routines that rely on high-speed interrupt capability.

- Interrupt routines should patch in as late as possible in the interrupt process without losing data. If your interrupt source doesn't need servicing as fast as AppleTalk does, don't patch in before AppleTalk.

- Patching the main interrupt vector at $E10010 is **only** acceptable if there's **no** possible way to service external interrupts quickly enough (internal interrupt sources, like serial ports, should always use other vectors), and even then the vector most only be patched while necessary; if a slower interrupt source is used in the same program, unpatch the vector.

- Different methods of installing interrupt handlers give you different levels of control. `BindInt` is the overall best method, although you can't control in what order bound routines are called.

- `COP` instructions are unacceptable in non-debugging code; they should never take the place of `JSL` instructions or other methods of inter-process communication.

Further Reference

- *Apple IIGS Firmware Reference*
- *Apple IIGS Toolbox Reference*, Volume 3
- *GS/OS Reference*
- *ProDOS 8 Technical Reference Manual*
- Apple IIGS Technical Note #24, *Apple IIGS Toolbox Reference* updates

Apple II
Technical Notes

Developer Technical Support

Apple IIGS #106: ADB Addendum

Written by: Dave Lyons

May 1992

This Technical Note documents some bits in the ADB `SendInfo` data byte for `setModes` and `clearModes`.

`SendInfo` is documented in Volume 1 of *Apple IIGS Toolbox Reference*, but it doesn't tell you what any of the bits in the `setModes`/`clearModes` data byte are for. Well, here are the useful ones:

Bit	Value	Description
6	$40	Shift+CapsLock=Lowercase mode
4	$10	Keyboard buffering
3	$08	Dual-speed keys
2	$04	Fast space/delete keys

For example, to turn off keyboard buffering without altering the user's Battery RAM, you can do the following:

```
             pea 1                    ;number of data bytes
             pushlong #modesToClear   ;pointer to data byte
             pea 5                    ;modeCmd = clearModes
             _SendInfo
             ...

modesToClear dc.b $10                 ;bit 4 = keyboard buffering
```

Note that the user's control panel setting will become current again if they hit Command-Ctrl-Esc (the system calls the `TOBRAMSETUP` vector at $E10094 to update the system from Battery RAM).

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 1

Developer Technical Support

Apple IIGS
#107: Tool Locator Tribulations

Written by: Dave Lyons May 1992

This Technical Note tells you what to watch out for in the Tool Locator.

ShutDownTools and System 6.0

In System 6.0, `ShutDownTools` inappropriately calls `HideCursor` even if QuickDraw II is not started. The results are unpredictable.

If your application does **not** use QuickDraw II but does use `ShutDownTools`, you may need to start up and shut down your tools manually instead.

Note that the `HideCursor` problem does not occur in the (unusual) case that the System 6.0 `noResourceMgr` bit (value $0010) is set.

Contents of the `StartUpTools` Tool Table

You should not include the Tool Locator or Memory Manager in your tool table. Instead, call `TLStartUp` and `MMStartUp` before calling `StartUpTools`, and call `MMShutDown` and `TLShutDown` after `ShutDownTools`.

Since `StartUpTools` automatically starts the Resource Manager for you, you should not include Resource Manager in the tool table, either. Doing so has no ill effect in System Software 6.0 and later, but in System Software 5.0 through 5.0.4 you got duplicate `ResourceStartUp` and `ResourceShutDown` calls.

The order of the tool table entries does not matter. (Toolbox Reference 3, page 51-8, says "Although `StartUpTools` handles the order of tool startup for you...", but this is widely overlooked.)

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 3

Apple IIGS
#108: Finder Funkiness

Written by: Dave Lyons May 1992

This Technical Note tells you what to watch out for in Finder 6.0.

Icon Search Order

When the Finder looks for an icon it uses the first match it finds. When more than one icon would match, the order is important.

Some icons are built right into the Finder's resource fork—those are always searched **last**. Other than that, the Finder searches in device-number order (for example, icons on device number $0001, the boot device, override icons on other devices).

On each disk, icons in `Desktop` files override icons in old-style icon files. Among old-style icon files in the same Icons folder, each icon file overrides subsequent (as returned by `GetDirEntry`) files in the same directory. Within an icons file, earlier icons override later icons.

If you create a "generic" icon that matches a broad class of files—for example, all files of a particular file type—you have to be very careful where you put that icon. A generic icon in any file's `rBundle` will wind up in a `Desktop` file, where it will override some old-style icon files (or all of them, if the `Desktop` file is on the boot disk).

There's really no good place for a custom generic icon. (Well, the Finder's resource fork would be a good place, but we recommend not messing with that.) A halfway-good place is in old-style icons folders, at the end, on the highest-numbered convenient device (for example, your third hard drive partition of three).

Note that the 6.0 Finder's matching order for old-style icons is more or less the reverse of what it was in previous versions.

Filename Matching and Wildcards

When an icon matches by filename and has a leading wildcard, the match always fails if there are any lowercase characters in the string. For example, "`*.TXT`" is fine, but "`*.Txt`" never matches.

Also, a leading wildcard matches one or more characters, instead of (as intended) zero or more characters. "`*ICONS`" matches "`MyIcons`" and "`Other.Icons`", but not "Icons". You can usually work around this by omitting the character after the wildcards: "`*CONS`" matches all three.

These notes apply both to old-style icon files and to new `matchFilename` structures.

Shut Down Default is Not Configurable

The System 6.0 Finder Documentation shows one of the words in the `rRectList`(1) resource as the default choice for the Shut Down dialog. Actually, the default is not configurable, and this word in the resource should remain zero. Utilities can customize the Finder's "Shut Down…" command by accepting the `finderSaysMItemSelected` request.

Further Reference

- System 6.0 Documentation