

# Apple IIGS
# #50: Extended Serial Interface Error Handling

Written by: Dan Strnad January 1989

This Technical Note discusses error reporting by the Extended Serial Interface.

---

For Apple IIGS ROM 01, the Extended Serial Interface does not return the error condition in the carry bit. Programs using the Extended Serial Interface should check for a non-zero result value in the **result code** rather than the carry bit to determine if an error has occurred. The following eight-bit APW code demonstrates this error checking using the `SetDTR` command. The `SetDTR` routine zeros the result bytes if no error has occurred.

```
            LONGA   OFF             ;PREPARE ASSEMBLER FOR EMULATION MODE
            LONGI   OFF
            65C02   ON
            KEEP      SETDTR2
            START
SLOT        EQU     $01
            SEC                     ;SET EMULATION MODE
            XCE
            JMP     BEGIN
CMDLST      DC      H'03'           ;PARAMETER COUNT
            DC      H'0B'           ;SETDTR COMMAND CODE
RESLT       DC      I'0'            ;RESULT CODE (OUTPUT)
DTRSTAT     DC      I'0'            ;BIT 7 IS STATE OF DTR (INPUT)
BEGIN       LDA     #SLOT           ;COMPUTE $CN VALUE TO BE USED
            ORA     #$C0
            STA     OFFSET+2        ;MODIFY INSTRUCTIONS LOADING OFFSETS
            STA     XOFFSET+2
            STA     ICALL+2         ;MODIFY INSTRUCTIONS CALLING FIRMWARE
            STA     XCALL+2
IOFFSET     LDA     $C00D           ;THIS INSTRUCTION MODIFIED AT RUNTIME
            STA     ICALL+1         ;MODIFY JSR TO INIT
XOFFSET     LDA     $C012           ;THIS INSTRUCTION MODIFIED AT RUNTIME
            STA     XCALL+1         ;MODIFY JSR TO EXTENDED SERIAL INTERFACE
ICALL       JSR     $C000           ;THIS INSTRUCTION MODIFIED AT RUNTIME
            LDA     #<CMDLST        ;LOW BYTE OF COMMAND LIST
            LDX     #>CMDLST        ;HIGH BYTE OF COMMAND LIST
            LDY     #0              ;24-BIT ADDRESS NOT USED BY 8-BIT PROGRAM
XCALL       JSR     $C000           ;THIS INSTRUCTION MODIFIED AT RUNTIME
            LDA     RESLT           ;DID AN ERROR OCCUR?
            BNE     ERROR           ;YES- HANDLE THE ERROR
            ...
ERROR
            ...
            END
```

Apple II
Technical Notes

Developer Technical Support

# Apple IIGS
# #51: How to Avoid Running Out of Memory

Revised by: Dave Lyons May 1992
Written by: Eric Soldan January 1989

This Technical Note discusses handling nearly-out-of-memory situations when working with the IIGS tools.
**Changes since September 1990:** Added discussion of an Out-of-memory routine problem fixed in System 6.0.

## Introduction

Running out of memory is a concern for most every application. Working with the Toolbox makes monitoring this situation a little more difficult since your application is not the only one allocating memory.

Low-level toolbox functions (for example, QuickDraw II calls) require that a 16K block of memory be allocatable, while high-level routines (for example, the Window Manager) require that a 32K block of memory be allocatable. Apple does not guarantee that toolbox functions behave reasonably if there is less memory available, and the tools are not stress-tested with less than the minimum required memory available.

Since the toolbox assumes reasonable memory-allocation requests succeed, just waiting for an out-of-memory error is **not** adequate memory management. To make your application work reliably in low-memory situations, you need a method of ensuring that the toolbox gets memory when it needs it. This Note describes two approaches.

## How Much Memory Can Be Allocated

There's no way to tell how much memory can be allocated without actually trying to allocate it.

`MaxBlock` tells you the size of the largest single free block, but this doesn't take into account purgeable blocks, compaction, and out-of-memory routines (see *Apple IIGS Toolbox Reference*, volume 3). `FreeMem` and `RealFreeMem` cannot tell you how badly fragmented the memory is, and they do not take into account out-of-memory routines.

## A Suggested Method

A method of checking for a nearly-out-of-memory condition is to have your own purgeable handle just for this task. If the handle has not been purged, then you have plenty of memory for the toolbox, and in the worst case, the toolbox purges your handle if it needs the RAM.

The less often your purgeable handle gets purged, the better performance you get in nearly-out-of-memory situations. Therefore, you should arrange for other purgeable memory, not necessarily belonging to your application, to be purged **before** your handle. For example, you want dormant applications to be purged, rather than having your handle get repeatedly purged and reallocated. So the purge level of this handle should be one.

The check to see if a handle has been purged is very fast. If it has been purged, you have to try to reallocate it. Reallocating a handle is not a fast process, so the fewer times the handle is purged, the faster the check is and the better your performance. Unless you are in a nearly-out-of-memory situation, the handle should not be purged at all, and you should have virtually no overhead for this process.

This technique can be implemented as follows:

```
appStart
;
; Somewhere at start, create a purgeable handle of size N,
; called "loMemHndl", purge level 1.
;
                rts

******************
;
; Here's an example of checking for nearly-out-of-memory:
;
                jsr    preCheckLoMem
                bcc    goForIt
                bcs    HandleError        ;Handle errors appropriately.
goForIt         (_ToolboxCall[s])         ;Make as many as needed.
;
; Here you can make your toolbox calls.  Since you prechecked
; for nearly-out-of-memory conditions, you should have no memory
; errors at this point.
;
; You could also check after calls, as shown here:
;
                (_ToolboxCall)
                jsr    checkLoMem         ;Call this to see if low.
                bcc    noError
                bcs    HandleError        ;Take care of errors.

noError         jsr    lifeIsGood
                .
                .
                .
                rts
```

```
******************
;
; Here are some sample routines to check for the nearly-out-of-
; memory condition.
;
checkLoMem      bcs    retErr
preCheckLoMem   lda    [loMemHndl]
                ldy    #2
                ora    [loMemHndl],y
                beq    gotPurged
                lda    #0
                clc
                rts
gotPurged       (Try reallocating it into loMemHndl, purge level 1.)
                (If you can't, you will get a $0201 error.  You may wish to
                 return the $201 error, or you may wish to change it into
                 your own error code.)
;
retErr          rts                     ;This is a single exit point
                                        ;whether errors were present
                                        ;or not.
```

You can determine the size of this purgeable handle, but like determining what size stack is adequate for an application, there is no single "right" answer. There are different considerations for size of the purgeable handle for each application, and these may change during the development process. Use your best judgement, keeping in mind that high-level toolbox routines require a 32K block.

## An Alternative

For better control over when your handle is purged or disposed, you can write an out-of-memory routine as described in the Memory Manager chapter of *Apple IIGS Toolbox Reference*, volume 3. Out-of-memory routines have the opportunity to free up memory before or after the Memory Manager attempts to purge purgeable handles, and this manual contains a sample of such a routine.

**Note:** If your Out-of-memory routine frees up memory on the second pass, there is a problem with the Memory Manager in System Software 5.0 through 5.0.4 that may affect you. If your routine frees enough bytes on the second pass, but the Memory Manager still cannot complete the request it is working on, it can hang for a couple of minutes and then crash. This is fixed in System 6.0.

**Further Reference:**

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



# Apple IIGS
# #52: Loading and Special Memory

Revised by: Dave Lyons May 1992
Written by: Eric Soldan January 1989

This Technical Note discusses strategies for preventing applications from loading into special memory.
**Changes since July 1989:** Noted that the System 6 Loader always tries non-special memory before special memory.

---

The System 6.0 Loader always tries to load segments into non-special memory before allowing them to load into special memory. The rest of this Note is useful if your application does not require System 6, or if you need an example of an initialization segment.

The System Loader loads your application starting at the lowest memory location possible. If you allow your program to load into special memory, the Loader first tries bank $01. If your program cannot load into special memory, it starts at bank $02. Either way, the Loader progresses to higher banks, and eventually, it may even try loading into bank $E1, which contains the super hi-res screen.

The problem with allowing your application to load into special memory is that the super hi-res screen is part of special memory. If you have a desktop application, part of your application may load into the super hi-res screen, and when you try to start QuickDraw II, it fails because the screen memory is already allocated.

When QuickDraw II fails because your program loaded into the SHR screen, it seems reasonable to assume that the Loader put your program there because it needed the RAM which special memory provides. This logic seems to make sense, but it is not completely reliable. The Loader (in System Software earlier than 6.0) tries to put your program into special memory **before** it tries purging dormant applications. This means that the more programs that run from the Finder that set the GS/OS or ProDOS 16 "restartable from memory" bit, the more likely it is that the next application launched that can load into special memory will load into the super hi-res screen.

For this reason, it is important not to let your application load into special memory, or at least not load into the super hi-res screen. If your application is not allowed to load into special memory, then the Loader will purge other dormant applications to make space for yours. One way to accomplish this is when linking your application. You can set the "no special memory" bit in the OMF `KIND` field of applications using OMF 2.0 or later, but this also prohibits your application from using bank $01.

Another way to avoid loading into the super hi-res screen is to have your initial segment allocate the super hi-res screen. You can accomplish this by starting QuickDraw II in your initial segment, then the rest of your program cannot load into the already-allocated super hi-res screen. This strategy could fail if the initial segment loaded into the super hi-res screen, but this is very unlikely and can be prevented by flagging the initial segment to only load into non-special memory. You can do this by setting the "no special memory" bit in the `KIND` field only for the initial segment.

Here's an example of such an initial segment in MPW IIGS format:

```
*********************************************************************
*
* You may wish to do this stuff in the initial segment of your
* application.  The initial segment should be set so that it does not
* load into special memory, or else it is possible that it would load
* into the super hi-res screen.  If this occurred, then QuickDraw II would
* not be able to be started.
*
* Once QuickDraw II is started, the super hi-res screen is taken,
* therefore the rest of the application can not load into it.  Therefore,
* special memory is generally an acceptable place for the rest of the
* application to load, since the special memory needed for the screen
* is already taken.
*
* If the performance of your application would be adversely affected
* by memory fragmentation, then you should also consider purging
* other dormant applications and dormant tools, and then compacting
* memory.  This will prevent fragmentation as much as possible
* while your application is loading.  It also has the cost of longer
* startup time since some tools may have to be reloaded.  This is the
* only way to be sure that tools that you don't want are removed
* from memory before the rest of your application tries to load
* around them.
*
* The Finder is a dormant application when your application is
* launched.  This will cause the Finder to be thrown out of memory,
* and it will have to be reloaded when your application is quit.
*
*********************************************************************

                    case on

                    include 'e16.memory'
                    include 'm16.memory'
                    include 'm16.quickdraw'

screenMode          equ     $80
AppMaxWidth         equ     160             ;Double this and your application
                                            ;will print in BetterText mode.
******************

initialScreen  PROC

myID                equ     1               ;long
zpagehndl           equ     myID+4          ;long

stkAfterLocals equ  zpagehndl+4

directReg           equ     stkAfterLocals
retAddr             equ     directReg+2
passedParms         equ     retAddr+3

                    phd                     ;Set up stack frame.
                    tsc
                    sec
                    sbc     #stkAfterLocals-1
                    tcs
                    tcd
                    pha
                    _MMStartUp
                    pla
                    sta     myID            ;Get the userID
```

```
                    pha
                    _HLockAll           ;Lock down the rest of ourselves, in
                                        ;case we are being restarted.  The
                                        ;loader does not prelock down stuff,
                                        ;so we would be disposing of the rest
                                        ;of ourselves.

                    pea    $1000
                    _PurgeAll           ;Purge other dormant applications.
                                        ;This is optional.
                    pea    $4000
                    _PurgeAll           ;Purge dormant tools.
                                        ;This is optional.

                    _CompactMem         ;Clean up memory.  This is advised.

                    pha                 ;Make direct space for QuickDraw.
                    pha
                    pea    $300>>16     ;Hi-byte of $300 address.
                    pea    $300
                    pei    myID
                    pea    attrLocked+attrFixed+attrPage+attrBank
                    lda    #0
                    pha
                    pha
                    _NewHandle
                    plx
                    stx    zpagehndl
                    plx
                    stx    zpagehndl+2
                    bcc    @a
                    ERRORDEATH 'Out of bank 0 memory'

@a                  lda    zpagehndl
                    sta    >qdstarthndl ;Used for disposing handle at shutdown.
                    txa
                    sta    >qdstarthndl+2
                    lda    [zpagehndl]  ;Start up QuickDraw.  This protects
                    pha                 ;screen RAM from the rest of the
                    pea    screenMode   ;application loading into it.
                    pea    AppMaxWidth
                    pei    myID
                    _QDStartUp
                    bcc    @b
                    ERRORDEATH 'Can''t start up QuickDraw'
@b                                      ;Do title screen here.
                    tsc
                    clc
                    adc    #stkAfterLocals-1
                    tcs
                    pld
                    rtl

qdstarthndl         dc.l   0

                    ENDP
                    END
```

## Further Reference:

- *GS/OS Reference*, Volume 1
- *MPW IIGS Tools Reference*
- *APW Assembler Reference*



# Apple IIGS
# #53: Desk Accessories and Tools

Revised by: Dave "Out of Phase" Lyons May 1992
Written by: Matt Deatherage, Jim Mensch, & Dave Lyons March 1989

This Technical Note discusses compatibility issues that can arise between desk accessories and applications. Where possible, it presents solutions.
**Changes since March 1991**: Updated information about QuickDraw Auxiliary and `StartUpTools` for System 6.0.

---

This Note presents guidelines to help applications and desk accessories work together smoothly.

## Tool Sets

The greatest conflict between applications and desk accessories, especially NDAs, is the use of system tool sets. The *Apple IIGS Toolbox Reference*, Volume 1, defines the minimum collection of tools sets available to an NDA. The Desk Manager requires that an application start the following tool sets before calling `DeskStartUp`:

Tool Locator (#1)
Memory Manager (#2)
Miscellaneous Tools (#3)
QuickDraw II (#4)
Event Manager (#6)
Window Manager (#14)
Menu Manager (#15)
Control Manager (#16)
LineEdit (#20)
Dialog Manager (#21)
Scrap Manager (#22)

NDAs may assume that these tools are all present and running, so they do not need to check for their presence. NDAs can also use the following tool sets without special consideration for starting them up: Desk Manager, Scheduler, Apple Desktop Bus, and Integer Math.

In addition to the tool sets applications must start to support NDAs, Apple recommends that applications start the following tools:

QuickDraw Auxiliary (#18) (see discussion under QuickDraw Auxiliary)
Font Manager (#27)

These two additional tools are so widely used by desk accessories that they should be present. NDAs may not **assume** their presence, but it is reasonable to write an NDA that **checks** for them, with the assumption that they usually turn out to be available.

## NDA Guidelines

### Which Tool Sets Can An NDA Use?

- In general, NDAs can use the tool sets which have already been started up by the host application, even tools that are not guaranteed to be started up. Using other tool sets is trickier (see below).

- In general, NDAs should not start up tools that are already started up. (The Resource Manager is an exception.)

- The Resource Manager must be started separately by each client. See Apple IIGS Technical Note #71 for detailed information on using the Resource Manager from an NDA.

- Sound tools are an exception to the rule of freely using a tool which is already started. See the section "Sound Tools" sections later in this note.

- Some tool sets are easily started up each time they are needed, if they are not already present.

  Standard File is an excellent example. If an NDA needs to use Standard File, it should check to see if the tool is already running. If it is not running, the NDA must use `LoadOneTool` to load it, then it must allocate a page of direct-page space and start the tool before using it. When finished with the tool, the NDA must shut it down, deallocate the direct-page space, and unload it with `UnloadOneTool`. (A tool is already running if its `xxxStatus` function returns `TRUE` and does not return an error.)

  The important thing here is that the NDA shuts down Standard File immediately after using it, if it was not already started. This does not cause conflicts with the host application or with other NDAs.

  Note that by pre-initializing the result space of an `xxxStatus` call to zero, you can avoid caring whether the tool is present but not started or simply not present.

```
pea $0000
_SFStatus
pla               ;A is nonzero if Standard File is started
```

  From a high-level language, you may not be able to pre-initialize the result space. Instead, you need something like the C statement:

```
StdFileActive = ( SFStatus() && !_toolErr);
```

  or the Pascal statement:

```
StdFileActive := (SFStatus<>0) AND (ToolErrorNum=0);
```

- It is impractical or impossible to start up certain tool sets each time they are needed. These include the Font Manager, Scrap Manager, and Text Edit.

  If an NDA needs to start up a tool and **keep** it started while letting the application continue to run, things get interesting. (There is a risk that the host application will later try to start up the tool set itself and not be able to deal with the tool already being started.)

  In practice, the safest thing you can do for a tool you need to leave running is:

—When your NDA is opened, check the tool set's status. If it is not available, use `LoadOneTool`, allocate any needed direct-page space, start up the tool set, and set a flag indicating that your NDA started the tool set.

—When your NDA's Init routine is called at `DeskShutDown` time (Accumulator equal to zero), check the flag set above. If your NDA started a tool set, shut it down, dispose of any direct-page space you allocated for it, and call `UnloadOneTool`.

(Keep in mind that your NDA can be opened and closed many times before `DeskShutDown` is called when the application finally quits. If you have started a tool and set a flag on an open, be sure not to disturb the flag on a future open, when the tool is already available because you started it! You still need to shut it down at `DeskShutDown` time.)

—Do **not** shut down tool sets when your NDA is closed. To see why, consider what would happen if two NDAs just like yours were used at the same time. If the NDAs were closed in any other than the exact opposite order they were opened, some NDAs would have tool sets shut down from underneath them.

### StartUpTools

- `StartUpTools` in System Software 5.0.4 and earlier is designed to be called only by an **application**, not a desk accessory. Unexpected things happen if your NDA calls `StartUpTools` (for example, you may get a second copy of the application's resource fork open in your NDA's private resource search path; this wastes RAM and can interfere with an application's attempt to write to its own resource fork).

- See the System 6.0 Toolbox documentation for information on using `StartUpTools` from an NDA. There are new flag bits you need to know about.

### TLStartUp and TLShutDown

- Do not call `TLStartUp` or `TLShutDown` from a desk accessory.

- You may call `MMStartUp` at any time to get your desk accessory's own memory ID. This does not allocate a new ID; it just tells you what ID you already have (it returns the memory ID of the block the `MMStartUp` call is made from).

### User Tool Sets Belong to the Application

- A desk accessory must not install user tool sets, because there is no arbitration of user tool set numbers. User tool sets are the sole property of the current application.

  A desk accessory should not call user tool sets even if it determines that the host application has installed a certain tool set, because that limits future system software options. For example, consider a hypothetical multiple-application environment. If DAs call user tool sets and the system automatically switches between separate collections of user tool sets, there would be no way for the system to know which set to switch in before giving control to a desk accessory.

### Bank Zero Memory and Error $0201

- If you get error $0201 (unable to allocate memory block) while trying to launch a ProDOS 8 application, it is probably because your NDA allocated some memory in bank 0 or bank 1 and failed to dispose of it at DeskShutDown time (when the NDA's Init routine is called with the accumulator equal to zero). GS/OS needs to allocate all of this memory for ProDOS 8 to use.

### QuickDraw Auxiliary

- In System 6.0 and later, QuickDraw Auxiliary is always available to an NDA, because the Window Manager automatically loads and starts QuickDraw Auxiliary (because it's needed for `AlertWindow`, for example). To prevent problems, duplicate `QDAuxStartUp` and `QDAuxShutDown` calls are tolerated, and `QDShutDown` automatically calls `QDAuxShutDown`.

- Before System 6.0, starting QuickDraw Auxiliary when the application has not started it can be a problem. An application that correctly implements switching between 320 and 640 mode calls `QDShutDown` and `QDStartUp`. QuickDraw Auxiliary depends heavily on QuickDraw, and restarting QuickDraw while QuickDraw Auxiliary is active will fry big-time.

### Sound Tools

- A desk accessory **cannot** use any of the sound tools if they are already started. This is contrary to the rule for other tool sets, but it is required because there is no memory management of the sound RAM (or "DOC RAM"). If the Sound Tools (#8) are started, the application has exclusive control of the 64K DOC RAM used to play sounds. Anything your desk accessory might put there could overwrite information the application needs.

  Saving and restoring DOC RAM around desk accessory usage is **not** sufficient. Many of the sound functions are interrupt driven, altering the contents of DOC RAM only during sound interrupts, so your desk accessory might attempt to replace parts of DOC RAM which are being played. Since there is no memory management of DOC RAM, desk accessories must avoid the sound functions of the IIGS if the application is already using them.

## Application Guidelines

For best compatibility with NDAs, applications should follow the following guidelines.

- Be careful about when your application starts and shuts down tools. A highly compatible approach is to start tools at the beginning of your application and leave them started. For certain tools, like Standard File, it is reasonable to load and start the tool set each time it's needed (you may want to check whether it's already started, in case some impolite NDA started Standard File and left it started).

  Note that `UnloadOneTool` followed later by `LoadOneTool` does not necessarily cause disk access or ask the user to insert the boot disk. `UnloadOneTool` calls `UserShutDown` to put the tool set into "zombie" state, where it can be restarted from memory if none of its segments have been purged. Unloading tools while they aren't in use is a Good Thing—if the user has plenty of RAM, there's no noticeable performance hit, but

if RAM space is tight then doing extra disk access still is preferable to actually running out of memory.

For maximum compatibility, an application should not shut down any tools which were ever active when it called `SystemTask` or `TaskMaster` (until quitting time, of course, when it shuts down everything, starting with the Desk Manager). The application can start more tools, but it should not shut down those which are already active.

If your application is going to start a tool and not keep it started, use it and then shut it down immediately, without allowing desk accessories to be opened during that time.

- Don't just start the Scrap Manager—use it! Many desk accessories support cutting and pasting to exchange text and pictures with your application, but they can do it only if you use the Scrap Manager. If you have a need for your own private scrap internally, you should still **also** use the Scrap Manager to exchange text and pictures with other applications and DAs.

- Provide an Edit menu, and when an NDA window comes to the front enable the menu and the Undo, Cut, Copy, Paste, and Clear items.

- Applications should never make a `Close` call with reference number zero at file level zero. (If you need to use `Close` with reference number zero, use `GetLevel` and `SetLevel` to avoid closing files you did not open.)

  DAs written recently can open their files at an internal file level, as documented in GS/OS Technical Note #13, but applications still need to avoid closing all files at level zero for compatibility with older desk accessories.

- An application with some memory to spare can save NDAs time by providing them the additional tools which they are most likely to use.

  The most common tools which desk accessories require besides those available in the standard Desk Manager set are QuickDraw Auxiliary (#18), the Print Manager (#19), Standard File (#23), the Font Manager (#27), and the List Manager (#28).

- When you call `TaskMaster` or `GetNextEvent`, or `EventAvail`, be sure bit 10 is turned on in the event mask, to enable "desk accessory" events. If you turn this bit off, users will not be able to get to the Classic Desk Accessory menu by pressing Apple-Ctrl-ESC.

## CDA Guidelines

- CDAs are nearly always modal, but by using the HeartBeat interrupt queue or other mechanisms, they can get control when the user is no longer "in" the CDA. The list of guaranteed tools for NDAs does not apply to CDAs, and CDAs must be prepared to deal with the ProDOS 8 environment as well as GS/OS.

- Under ProDOS 8, a CDA will not be able to allocate any bank 0 space through the Memory Manager; it can only use page 0 and page 1 safely (the stack is in page 1).

- Do not call `TLStartUp` or `TLShutDown` from a desk accessory.

- You may call `MMStartUp` at any time to get your DA's own memory ID. This does not allocate a new ID; it just tells you what ID you already have (it returns the memory ID of the block the `MMStartUp` call is made from).

**Further Reference:**

- *Apple IIGS Toolbox Reference*
- *Programmer's Introduction to the Apple IIGS*
- Apple IIGS Technical Note #71, Desk Accessory Tips and Techniques
- Apple IIGS Technical Note #83, Resource Manager Stuff



# Apple IIGS
# #54: MIDI Drivers

Revised by: Matt Deatherage November 1990
Written by: Jim Mensch May 1989

This Technical Note describes how to write a driver for use with the Apple IIGS MIDI tools.
**Changes since May 1989:** Noted that MIDI drivers also work with the MIDI Synth tool.

---

Apple ships two drivers with the MIDI tool set, `APPLE.MIDI` and `CARD6850.MIDI`, respectively. These drivers are adequate for almost all MIDI hardware currently on the market for the Apple IIGS; however, if your hardware is not compatible with either of these drivers, you have to write your own. This Note includes all the information you need to create a MIDI driver. Note that the same drivers that work with MIDI Tools (Tool #32) also work with the MIDI Synth (Tool #35). This Note collectively refers to MIDI Tools and MIDI Synth as the "MIDI tools."

## Purpose of the Driver and Description of Hardware Requirements

The Apple MIDI tools communicate to the MIDI world via a simple driver. The driver's function is managing the transmission and reception of single bytes of MIDI data between the tools and the particular MIDI hardware involved. The MIDI tools operate on the assumption that the hardware has a method of interrupting the system when a character has been received and when a character can be transmitted. Since there is quite a bit of overhead in processing MIDI data, and since MIDI data can comes across a standard MIDI bus at a rate of over 3000 bytes per second, it is suggested that you provide a means for your device to buffer a few characters to reduce system overhead caused by interrupts if you are designing hardware to be used with the MIDI tools.

## Format of the Driver File

The driver file is a standard OMF load file, which can be created with any of the popular Apple IIGS assemblers. The file must start with a dispatch table that contains the addresses of the standard driver routines. All driver routines must be in the same segment as the dispatch table. The dispatch table should have 13 four-byte entries, each of which contains the address of the appropriate routine minus one. Table 1 contains addresses of routines in the MIDI driver to perform specific functions.

| Call | Function |
|---|---|
| `Init` | Called to initialize the port and prime the driver |
| `ShutDown` | Called to close the port and clean up after driver |
| `Reset` | Called at reset time by the MIDI tools |
| `IntHandler` | Called when your interrupt occurs |
| `PollRecv` | Poll input the port for data |
| `RecvIntOn` | Turns on receiver interrupts |
| `RecvIntOff` | Turns off receiver interrupts |
| `PollXmit` | Polls the transmitter to see if another character can be sent |
| `XmitIntOn` | Enables transmitter interrupts |
| `XmitIntOff` | Disables transmitter interrupts |
| `NotImp` | Currently unused |
| `NotImp` | Currently unused |
| `NotImp` | Currently unused |

**Table 1–MIDI Driver Function Routines**

## Routine Calling Conventions

All driver routines are called with full 16-bit mode enabled and should exit the same way. On entry to each routine, the accumulator contains the direct page pointer that the driver should use if it wants to use the MIDI Tools' or MIDI Synth's direct page. It is the driver's responsibility to set the direct page register and restore it on exit. All other parameters are passed on the stack and should be removed from the stack before the routine exits. The MIDI tools set aside 128 bytes of space on the passed direct page for use by the driver. They are bytes $80–$FF.

If you want to report an error inside of any routine (except `IntHandler`), set the carry flag on exit and load the accumulator with the error code. Use predefined error codes whenever possible. If you need to report a device specific error, use errors in the range $C0–$FF. The MIDI tools will set the high byte of the error code properly for you, so you do not need to do it yourself. Table 2 lists all of the potential predefined error codes.

| Error Code | Error Definition |
|---|---|
| `miToolsErr` ($2004) | The required tools were not started |
| `miNoBufErr` ($2007) | No buffer is currently allocated |
| `miDevNotAvail` ($2080) | Requested device is not available |
| `miDevSlotBusy` ($2081) | Requested slot is already in use |
| `miDevBusy` ($2082) | Requested device is already in use |
| `miDevOverrun` ($2083) | Device overrun by incoming MIDI data |
| `miDevNoConnect` ($2084) | No connection to MIDI |
| `miDevReadErr` ($2085) | Framing error in received MIDI data |
| `miDevVersion` ($2086) | ROM version is incompatible with driver |

| | |
|---|---|
| `miDevIntHndlr` ($2087) | Conflicting interrupt handler installed |

## The Driver Routines

### Init

This routine is called by the MIDI tools when it wants to initialize your port and tell the driver to prepare itself for the rest of the calls. Figure 1 shows how the stack looks on entry to this call.

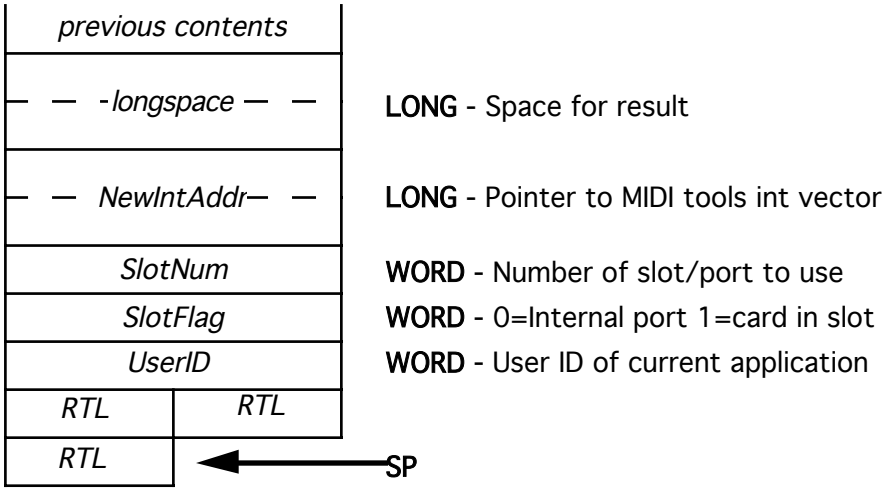


**Figure 1–The Stack on Entry to Init**

The `Init` routine should first test to see if the port specified by `SlotFlag` and `SlotNum` is available for use. `SlotNum` is the number of the slot or the port that the user has requested for use, and `SlotFlag` indicates whether it is a built-in port or a card in a slot. After determining that the requested device is available, you should initialize the device, allocate any memory that your driver may require (beyond what is available in the direct page), and set the proper system interrupt vector to the address passed in `NewIntAddr`. Before setting the vector, be sure to save the old value, as the MIDI tools expect the result from this routine to be the old address stored in the vector. On exit, the stack should contain the return address and the old vector address.

### ShutDown

This routine is called when the MIDI tools want your driver to release the MIDI device and prepare to be unloaded. Figure 2 shows how the stack looks on entry to this call.


**Figure 2–The Stack on Entry to ShutDown**

Your routine should change the interrupt vector that you used to `OldIntVector`. It should then deallocate all the memory that it allocated, disable all interrupts on the device, and if needed, tell the system that you are no longer using the port in question.

**Reset**

This routine is called when the system has been reset by the user. Figure 3 shows how the stack looks on entry to this call.


**Figure 3–The Stack on Entry to Reset**

All you should do at this point is attempt to deallocate any memory you were using and disable interrupts on the device you were using.

**Note:** Do not set the interrupt vector to `OldIntVector`, instead remove the value from the stack and dispose of it.

**IntHandler**

The `IntHandler` routine is called by the MIDI tools when an interrupt occurs for the vector that you are using. The MIDI driver performs some setup then calls your routine. This routine does not have any parameters on the stack.

Once called, your `IntHandler` routine should test the port to see if an interrupt has occurred on your device. If your device did not cause the interrupt, you should set the carry and exit as quickly as possible, reducing the system interrupt overhead.

If your device caused the interrupt, you should test the receiver to see if any bytes of data are waiting to be read. If there is data waiting, you should load that data into the accumulator and perform a `JSL` to the following code:

```
InBufGlue    PEA $0400
             PHD
             RTL
```

This code calls the MIDI tools and tell them to accept the character in the accumulator into its input buffer. After accepting the data, control is passed back to the instruction following your `JSL`. If you received a byte of data and an error occurred during reception, you should load the number of the error code into the `y` register and perform a `JSL` to the following code:

```
InErrGlue    PEA $0500
             PHD
             RTL
```

Again, you will regain control right after the `JSL`. Once in your interrupt routine, you may perform the calls above for as much data as you like. For example, if your device has a three-byte buffer, you could call `InBufGlue` once for each waiting character, thus reducing your interrupt overhead and possibly preventing unneeded interrupts.

If the transmitter on your device is ready to send data, you should perform a `JSL` to the following code:

```
OutBufGlue    PEA $8400
              PHD
              RTL
```

This routine will return with the carry set if no data is waiting to be transmitted or the carry clear if data is available. If data is waiting, the next character to send will be in the accumulator, and you should simply send it at that time. If no more data is available, you should disable transmitter interrupts and exit. The MIDI tools will re-enable transmitter interrupts the next time it has data to send.

**PollRecv**

The `PollRecv` (Poll Receive) routine is called by the MIDI tools every now and then to see if any data might be waiting to be read. There are no parameters on the stack for this call. Your driver should test to see if any data is available and transmit it all to the MIDI tools via the `InBufGlue` described in the `IntHandler` description.

**PollXmit**

The `PollXmit` (Poll Transmit) routine is called by the MIDI tools when any data is added to the MIDI output buffer. There are no parameters on the stack for this routine. Your driver should enable transmitter interrupts, test to see if it can send any data immediately, and if it can, call `OutBufGlue` as described int the `IntHandler` description to get data to send.

**XmitIntOn and RecvIntOn**

These routines are called when the MIDI tools want to explicitly enable transmitter or receiver interrupts. They have no parameters on the stack and should, when called, enable transmitter interrupts for `XmitIntOn` and receiver interrupts for `RecvIntOn`.

**XmitIntOff and RecvIntOff**

These routine are called when the MIDI tools want to explicitly disable transmitter or receiver interrupts. They have no parameters on the stack and should, when called, disable transmitter interrupts for `XmitIntOff` and receiver interrupts for `RecvIntOff`.

**NotImp**

These routines are not yet implemented, but your driver should be ready to handle a call to them. When called, they should clear the accumulator, clear the carry and perform an `RTL` back to the MIDI tools.

## A MIDI Driver Skeleton

You can use the following sample code as a basis for a MIDI driver. It is not a complete driver in itself, and you will need to add code where comments with asterisks (***) appear for it to be functional. This example is in MPW IIGS assembler format.

```
*****************************************************************************
* MIDI.DRVR.Aii
*
* (C)  Copyright Apple Computer, Inc. 1988
* All rights reserved.
*
* by Don Marsh & Jim Mensch
* 10/26/88
*
* This is a shell that can be used to create custom MIDI drivers for use with
* the Apple MIDI tool set. This shell is not functional, but can be used as a
* starting point for creating your own custom MIDI drivers.
*
* Files:    System Macros and equates
*
*
*
* Modification History:
*
* Version 1.0   Mensch
*
*      10/26/88
*
*      Create first draft
*
*****************************************************************************
       Include 'E16.MIDI'
       Include 'M16.MiscTool'
       Include 'E16.MiscTool'
       Include 'M16.util'

;
; Direct page usage Note:
; MIDI drivers may use the upper half ($80-$FF) of the MIDI direct page. When
; a MIDI driver routine is called the Accumulator will contain the direct page
; pointer for the MIDI tool set. If your driver requires more storage than
; 128 bytes, it will have to allocate them itself using the memory manager.

theuserID     equ $80             ; location to store the passed user ID
PortInUse     equ theuserID+2     ; storage for the port number in use
deref         equ PortInUse+2
Temp          equ Deref+4
              EJECT
```

```
*******************************************************************************
*
DispatchTable RECORD
*
* Description:      Every MIDI Driver must start with a driver dispatch table
*      that contains the entry point minus 1 of each of the
*      required entry points.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
              Import DRVRInit
              Import DRVRShutDown
              Import DRVRReset
              Import DRVRIntHandler
              Import DRVRPollRecv
              Import DRVRRecvIntOn
              Import DRVRRecvIntOff
              Import DRVRPollXmit
              Import DRVRXmitIntOn
              Import DRVRXmitIntOff
              Import DRVRNotImplemented
*
* Entry Points:     None
*
*******************************************************************************

              DC.L DRVRInit
              DC.L DRVRShutDown
              DC.L DRVRReset
              DC.L DRVRIntHandler
              DC.L DRVRPollRecv
              DC.L DRVRRecvIntOn
              DC.L DRVRRecvIntOff
              DC.L DRVRPollXmit
              DC.L DRVRXmitIntOn
              DC.L DRVRXmitIntOff
              DC.L DRVRNotImplemented
              DC.L DRVRNotImplemented
              DC.L DRVRNotImplemented


; a few of the routines will need a temporary storage location that can be used
; even after the direct page is set back to what it was, This is a good place
; to put it!

ErrorCode     ds.W 1              ; temporary holder of an error code
              EndR

              EJECT
```

```
*******************************************************************************
*
DRVRInit      PROC
*
* Description:      This is called by the MIDI Tools when it needs to Init
*      your MIDI Driver. This is usually in response to a MIDIxxx
*      call made by the application.
*      When this routine is called, you should allocate any buffer
*      space that you will need beyond the direct page, you should
*      enable the interrupts on your MIDI Device, and then set the
*      appropriate system interrupt vector and return the old vector
*      value. If the init works fine, clear the carry and return.
*      If an error occurs return the appropriate error code
*      in the Accumulator, and set the carry.
*
*
* Inputs:     UserID:Word         ID of application, for mem allocation
*             SlotFlag:Word       0 for internal port/ 1 for slot
*             SlotNum:Word        number of slot/port to use
*             NewIntVector:Long   address to give system as its new
*                                 interrupt vector. This routine is in the
*                                 MIDI tool set, and it performs needed
*                                 setup before it calls your interrupt
*                                 routine
*
* Outputs:    OldIntVector:Long   Address interrupt vector used to have
*
* External Refs:     None
*
* Entry Points:      None
*
*******************************************************************************
; Offsets for parameters on the stack

ProcStatus    equ 1
OldDPage      equ ProcStatus+1
ReturnAddress equ OldDPage+2
UserID        equ ReturnAddress+3
SlotFlag      equ UserID+2
SlotNum       equ SlotFlag+2
NewIntVector  equ SlotNum+2
OldIntVector  equ NewIntVector+4
ParmBytes     equ 10
ParmEnd       equ ReturnAddress+ParmBytes

; first disable interrupts since we are going to be setting up interrupt vectors
; and enabling interrupt generating hardware. We wouldn't want an interrupt to go
; off before we were ready to handle it! Then set us up to use the MIDI direct
; page.

              php                 ; save the old proc status
              phd                 ; save the old direct page
              tcd                 ; Set Direct page to the one passed
              SEI                 ; and disable interrupts

; now get the user ID and save it, and allocate any buffers that we may need
; Since most drivers will never need more than 128 bytes of storage we will
; not allocate any storage space

              lda UserID,s        ; first save the user ID for later
              sta theUserID       ; in our section of the MIDI DPage

; *** Insert any memory allocation needed here ***
```

```
; Next, you should check the slot flag and number to see if they are compatible
; with this driver. If they are, you should continue and initialize the proper
; port. If they are not proper, you should exit with an error.
; For this example, I will be testing the SlotFlag, to see if it is set to
; external.

             lda SlotFlag,s      ; first test the slot flag to be sure
             bne FlagOK          ; its non-zero.

             ldy #miDevNotAvail  ; if its zero, signal not available
             bra InitError       ; and exit via error routine

FlagOK       lda SlotNum,s       ; Now save the slot number in
             sta PortInUse       ; our data area

; *** At this point you should test the firmware in the desired slot to be sure
; that the card you want is properly installed, if it is not then you should
; pass back the appropriate error ***

; Now that you know that you have the proper slot information and you have tested
; to be sure that you have the hardware needed for the driver it is time for you
; to initialize the interface and to enable its interrupts.

; *** Install code to initialize your hardware/interrupts here ***

; Now that the Port has been properly initialized, you must set up the proper
; system interrupt vector. Since we required an external card above it would
; make sense that you need to use the "Other unspecified interrupt handler"
; vector (Number $0017). But first, remember to get the original vector pointer
; because we must return it to the MIDI tools.

             PushLong #0         ; space for result
             PushWord #otherIntHnd ; vector to retrieve
             _GetVector          ; and get the vector in question
             PullLong Temp       ; place in storage for a sec

             lda Temp            ; now place it on the stack
             sta OldIntVector    ; as the result of this function
             lda Temp+2
             sta OldIntVector+2

             lda NewIntVector    ; now move the MIDI Interrupt routine
             sta Temp            ; pointer into temporary storage
             lda NewIntVector+2
             sta Temp+2

             PushWord #otherIntHnd ; now set the vector to point to
             PushLong Temp       ; the MIDI drivers interrupt routine
             _SetVector

; The driver is now all set up, pull off the passed parms and we are done!
Done         ldy #0              ; set the error code to 0. No error
;
; This is the alternate label for the Done routine that should be called when
; an error has occurred.
InitError
             lda ReturnAddress,s ; Move the return address below the
             sta ParmEnd,s       ; parameters
             lda ReturnAddress+1,s
             sta ParmEnd+1,s

             pld                 ; get the direct page back
             plp                 ; get the processor status back
```

```
             tsc                 ; now adjust the stack pointer
             sec                 ; so that the parameters are gone
             sbc #ParmBytes
             tcs                 ; now the return address is on Top

             tya                 ; put any error into <A>
             cmp #1              ; set the carry if non-zero
             RTL                 ; and return

             EndP

             EJECT
*******************************************************************************
*
DRVRShutDown  PROC
*
* Description:      This routine will be called whenever the MIDI Tools want
*      to cause your driver to let go of the port it was using.
*
*
* Inputs:     OldIntVector:Long   Address to place back into the system
*                                 interrupt vector you were using
*
* Outputs:    Carry clear if successful
*             Carry set if not, error in <A>
*
* External Refs:
       Import DrvrRecvIntOff
       Import DRVRXMitIntOff
*
* Entry Points:
*
*******************************************************************************
             With DispatchTable

ProcStatus    equ 1
OldDPage      equ ProcStatus+1
ReturnAddress equ OldDPage+2
OldIntVector  equ ReturnAddress+3
ParmBytes     equ 4
ParmEnd       equ ReturnAddress+ParmBytes

; first disable interrupts since we are going to be setting up interrupt vectors
;  We wouldn't want an interrupt to go off before we were ready to handle it!
; Then set us up to use the MIDI direct page.

             php                 ; save the old proc status
             phd                 ; save the old direct page
             tcd                 ; Set Direct page to the one passed
             SEI                 ; and disable interrupts

             lda #0              ; zero out the temp error code
             sta >ErrorCode
; Now First, re-install the old interrupt vector

             lda OldIntVector    ; get the old vector off the stack
             sta Temp            ; and save it in globals for a sec
             lda OldIntVector+2
             sta Temp+2

             PushWord #otherIntHnd ; now set the vector to point to
             PushLong Temp       ; its original routine.
             _SetVector
```

```
; Next, turn off the interface hardware, and tell it to stop generating
; interrupts. We can share some code here and call our DRVRRecvIntOff and
; DRVRXmitIntOff routines. Always remember load the direct page into the
; accumulator.

             tdc                 ; get direct page into <A>
             jsl DRVRXmitIntOff  ; and turn off transmitter interrupts

             tdc
             jsl DRVRRecvIntOff  ; and now receiver interrupts.

; *** Usually turning off interrupts will be all that you would need to do at
; this point, however, if your interface card requires extra shutdown code
; this is where you would place it ***

; *** If you allocated any memory in the DRVRInit call, this is the place to
; get rid of it.

; If an error were to occur in this routine, you should simply store the error
; number in our temporary error code variable like this
;
;            lda #ErrorNumber
;            Sta >ErrorCode


Done
; Now that we are done shutting down the driver, pull off the passed data
; and end.
             pld                 ; first retrieve the old dpage
             plp                 ; and processor status

             Longa Off           ; next move the return address
             SEP #$20            ; we need a short acc for this trick

             pla                 ; pull the 3 byte return address
             ply                 ; into <A> and <Y>

             plx                 ; now remove the remaining bytes
             plx                 ; of passed parameters

             phy                 ; and restore the return address
             pha

             Longa On
             REP #$30            ; and turn back on full 16-bit mode

             lda >ErrorCode      ; retrieve the error code
             cmp #1              ; and set the carry if non-zero
             RTL
             EndP

             EJECT
```

```
*******************************************************************************
*
DRVRReset     PROC
*
* Description:      This routine will be called whenever MIDIReset is called.
*      and that should only happen when an actual reset occurred.
*      It should in most cases perform the exact same functions
*      as MIDI Shutdown.
*
*
* Inputs:     OldIntVector:Long   Original contents of interrupt vector
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

       jmp DRVRShutDown

       EndP

       EJECT
*******************************************************************************
*
DRVRIntHandler      PROC
*
* Description:      This routine is the very core of the MIDI driver. It takes
*      care of passing data back and forth between the MIDI tools
*      and your hardware. It will be called for both input and
*      output.
*
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:
             Import DRVRXmitIntOff
*
* Entry Points:
             Export InBufGlue
             Export InErrGlue
             Export OutBufGlue
*
*******************************************************************************

             phd                 ; first, save the current dpage
             tcd                 ; and use the MIDI DPage

; The first thing the interrupt routine should do is to test to see if the
; interrupt was actually generated by our port. If it was then we should handle
; it, but if not, we should simply exit this routine with the carry set as
; fast as we can, so that the next interrupt handler will get it in a timely
; manner.

; *** Insert code here to test to see if the original interrupt was yours ***

             beq ServicePort     ; if it was our, handle it

; If the interrupt was not ours, set the carry and leave
             pld                 ; restore the direct page
             sec
             rtl
```

```
ServicePort                     ; the interrupt was ours, continue

; This routine should test the interrupt again, too see if the port is ready
; to transmit or receive, If it is ready to transmit or receive, it should
; then call the ServiceRecv, or ServiceXMit routines

; *** Insert code here to test for receive

             bne ServiceRecv     ; if chars waiting try receive it

; If no more characters are waiting, see if we are ready to transmit any
; characters.

             bne ServiceXMit     ; if can send a character do it

; If both the above tests fail, then exit the interrupt handler for now
             pld                 ; restore the direct page
             clc                 ; clear the carry to indicate serviced
             RTL                 ; and return

; The following routine ServiceRecv will be called when a character is waiting
; It should retrieve that character, pass it to the MIDI drivers, and then
; branch back to the beginning of ServicePort, to see if any more chars are
; waiting.
ServiceRecv

; *** Place code here that retrieves a byte of data from the port ***

; Call MIDI tools this way if no error has occurred on receive (<A> contains the
; data read)
RecvOK
             jsl InBufGlue       ; call the MIDI tools
             bra ServicePort     ; and check for more data in or out

; Call MIDI this way if a reception error has occurred (<A> contains the
; data read)
RecvErr
             ldy #miDevReadErr   ; load Y with the error
             jsl InErrGlue       ; call the midi tools
             bra ServicePort

; The routine ServiceXmit will be called when the port is ready to send data.
; it will actually call the MIDI tools and get a character to send.
ServiceXmit

             jsl OutBufGlue      ; call the MIDI tools for the next char
             bcs NoMoreData      ; if the carry set then no data to send

; *** at this point the byte to transmit is in <A>, place your code to output
; it thru the port here ***


; Now that the data has been sent, you can either loop thru ServicePort again,
; or you could simply end and wait for the next interrupt to send another
; character. This sample will simply exit at this point
       bra Done      ; after sending the character end.
```

```
; NoMoreData is called when the MIDI Tools said that they did not have any more
; data to transmit, so we should turn off transmitter interrupts at this point
; in case our device likes to keep interrupting if its empty.
NoMoreData
             phd                 ; push the direct page reg on the stack
             jsl DRVRXmitIntOff  ; enable xmit interrupts
Done
             pld                 ; restore the DPage
             clc                 ; signal the interrupt as handled
             rtl                 ; and get outta here!

; The routine inbufglue should be called when you received a character from your
; port with no error and you want to pass it to the MIDI tools.
InBufGlue    pea $0400           ; push on the long address of the
             phd                 ; direct page and a proc status byte
             RTL                 ; and jump back to the MIDI tools

; The routine inErrGlue should be called when you received a character from your
; port and an error has occurred. In this case, it should still be passed to the
; MIDI driver, as it may still be useful
inErrGlue    pea $0500           ; push on the long address of the
             phd                 ; direct page and a proc status byte
             RTL                 ; and jump back to the MIDI tools

; The routine OutBufGlue should be called when you are ready to send a char
; out your port. The MIDI tools will will return with the character to send
; in <A>. If the MIDI tools have no more characters to send then OutBufGlue
; will return with the carry set.
OutBufGlue   pea $8400           ; push on the long address of the
             phd                 ; direct page and a proc status byte
             RTL                 ; and jump back to the MIDI tools
             EndP

             EJECT
*******************************************************************************
*
DRVRPollRecv  PROC
*
* Description:      This routine is called by the MIDI tools when it wants to
*      pool the port for data instead of waiting for an interrupt.
*      its function is similar to that of the our interrupt handler
*      except that it only does input.
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:
             Import InBufGlue
             Import InErrGlue
*
* Entry Points:     None
*
*******************************************************************************

             phd                 ; first, save the current dpage
             tcd                 ; and use the MIDI DPage
             php
             SEI
```

```
ServicePort                     ; the interrupt was ours, continue

; This routine should test the port too see if the port has any data for use
; to receive. If it does, it calls the MIDI tools and hands it off. Also note
; this routine will turn off interrupts, since we wouldn't want any stray
; receiver interrupts to spoil our fun and grab the data from us. (This is
; very important for certain types of ports which may signal that the port
; is ready and the generate an interrupt, thus leaving us in a situation where
; our interrupt routines could steal the interrupt right out from under us before
; we fetched it, thus allowing us to possibly double post a character.

; *** Insert code here to test for received data ***

             bne ServiceRecv     ; if chars waiting try receive it


; If no more data is waiting  exit this routine.
             plp
             pld                 ; restore the direct page
             clc                 ; clear the carry no errors possible
             RTL                 ; and return

; The following routine ServiceRecv will be called when a character is waiting
; It should retrieve that character, pass it to the MIDI drivers, and then
; branch back to the beginning of ServicePort, to see if any more chars are
; waiting.
ServiceRecv

; *** Place code here that retrieves a byte of data from the port ***

; Call MIDI tools this way if no error has occurred on receive (<A> contains the
; data read)
RecvOK
             jsl InBufGlue       ; call the MIDI tools
             bra ServicePort     ; and check for more data in or out

; Call MIDI this way if a reception error has occurred (<A> contains the
; data read)
RecvErr
             ldy #miDevReadErr   ; load Y with the error
             jsl InErrGlue       ; call the midi tools
             bra ServicePort
             EndP
             EJECT
```

```
*******************************************************************************
*
DRVRPollXMit  PROC
*
* Description:      This routine is called when the MIDI tools wants to start
*                   an output stream. The tool set calls this routine for the
*                   first character of data, and then this routine is
*                   responsible for enabling transmitter interrupts and sending
*                   the character.
*
*
* Inputs:     None
*
* Outputs:    Carry set if interrupt not serviced
*
* External Refs:    None
             Import OutBufGlue
             Import DRVRXmitIntOn
*
* Entry Points:     None
*
*******************************************************************************

             phd                 ; first, save the current dpage
             tcd                 ; and use the MIDI DPage
             php                 ; disable interrupts as we are now going
             SEI                 ; to turn on xmitter interrupts.

; First see if the port is ready to send any data, if not simply exit

; *** Insert code here to test if output is ready ***

             bcs Done            ; if not, then simply end

; The port is ready to accept a character for output so, call MIDI tools
; to get the next character

             jsl OutBufGlue      ; get the next character
             bcs Done            ; if carry set, no chars to xmit so end


             pha                 ; save the character to send
             phd                 ; push the direct page reg on the stack
             jsl DRVRXmitIntOn   ; enable xmit interrupts
             pla                 ; retrieve the character to send

; *** Insert code here to transmit a character ***
Done
             plp                 ; get the old interrupt status
             pld                 ; get the old direct page
             lda #0              ; no errors are possible
             clc
             rtl

             EndP

             EJECT
```

```
*******************************************************************************
*
DRVRXmitIntOn PROC
*
* Description:      This routine will be called when the MIDI tools need to
*      enable transmitter interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

              php                 ; save proc status/interrupt state
              phd                 ; save the old direct page
              tcd                 ; use the MIDI tools DPage
              SEI                 ; disable interrupts

; *** Insert code here to enable transmitter interrupts on your device

              pld                 ; recover old direct page
              plp                 ; recover old interrupt state
              lda #0              ; and return no-error (none possible)
              clc
              rtl
              EndP

*******************************************************************************
*
DRVRXmitIntOff      PROC
*
* Description:      This routine will be called when the MIDI tools need to
*                   Disable transmitter interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

              php                 ; save proc status/interrupt state
              phd                 ; save the old direct page
              tcd                 ; use the MIDI tools DPage
              SEI                 ; disable interrupts

; *** Insert code here to Disable transmitter interrupts on your device

              pld                 ; recover old direct page
              plp                 ; recover old interrupt state
              lda #0              ; and return no-error (none possible)
              clc
              rtl
              EndP

              EJECT
```

```
*******************************************************************************
*
DRVRRecvIntOn PROC
*
* Description:      This routine will be called when the MIDI tools need to
*      enable receiver interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

             php                 ; save proc status/interrupt state
             phd                 ; save the old direct page
             tcd                 ; use the MIDI tools DPage
             SEI                 ; disable interrupts

; *** Insert code here to enable receiver interrupts on your device

             pld                 ; recover old direct page
             plp                 ; recover old interrupt state
             lda #0              ; and return no-error (none possible)
             clc
             rtl
             EndP


*******************************************************************************
*
DRVRRecvIntOff      PROC
*
* Description:      This routine will be called when the MIDI tools need to
*      Disable receiver interrupts on your device.
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************

             php                 ; save proc status/interrupt state
             phd                 ; save the old direct page
             tcd                 ; use the MIDI tools DPage
             SEI                 ; disable interrupts

; *** Insert code here to Disable receiver interrupts on your device

             pld                 ; recover old direct page
             plp                 ; recover old interrupt state
             lda #0              ; and return no-error (none possible)
             clc
             rtl
             EndP
```

```
*******************************************************************************
*
DRVRNotImplemented  PROC
*
* Description:      Dummy routine, should leave the stack alone and return
*      no error
*
*
* Inputs:     None
*
* Outputs:    None
*
* External Refs:
*
* Entry Points:
*
*******************************************************************************
             lda #0
             clc
             RTL
             EndP

             END
```

## Further Reference:

- *Apple IIGS Toolbox Reference Update*



# Apple IIGS
# #55: Avoiding ClrHeartBeat

Written by: Matt Deatherage July 1989

This Technical Note lists changes to the description for `ClrHeartBeat`. This information supersedes the description in the *Apple IIGS Toolbox Reference Manual.*

---

The *Apple IIGS Toolbox Reference Manual* gives the following cautionary note in the description for the call `ClrHeartBeat`:

> *"A desk accessory may have installed tasks in the Heartbeat Interrupt Task queue. If you make a* `ClrHeartBeat` *call, you will remove those tasks. Therefore, under normal circumstances you should not make this call."*

This isn't rude enough to get the point across to some people, so we'll try again:

The Heartbeat Interrupt Task queue does **not** belong to the application. Different portions of System Software can, and will, install Heartbeat Tasks. If these tasks are removed, anything from a system crash to media corruption may result. **Nothing** but System Software should make this call.

## Further Reference

- *Apple IIGS Toolbox Reference Manual*



# Apple IIGS
# #56: Managing Dynamic Segments

Revised by: Matt Deatherage November 1990
Written by: Eric Soldan July 1989

This Technical Note discusses application difficulties when transferring control to dynamic segments during low-memory conditions.

**Changes since July 1989:** The information formerly covered in this Note is now discussed in greater detail in Apple IIGS Technical Note #22, Proper Use of Dynamic Segments.

---

This Note formerly warned of the dangers of using dynamic segments—if memory is not available for the dynamic segment, the system crashes. Apple IIGS Technical Note #22, Proper Use of Dynamic Segments, covers this problem and strategies for working around it.



# Apple IIGS
# #57: The Memory Manager and Interrupts

Revised by: Dave "nocturnal" Lyons December 1991
Written by: Dave Lyons July 1989

This Technical Note discusses how you can use the Memory Manager from interrupt routines and documents a flag byte that debugging utilities can use to temporarily prevent the Memory Manager from moving or purging memory.
**Changes since July 1989**: Expanded and retitled Note to discuss safe use of the Memory Manager at interrupt time.

---

The Memory Manager does not disable interrupts while it's busy. Instead, it increments the system `BUSY` flag when it's in the middle of something important.

## Can interrupt routines call the Memory Manager?

If you write code that executes at interrupt time, you must check the `BUSY` flag (the byte at $E100FF) before making any Memory Manager calls. If the `BUSY` flag is zero, it's safe to call the Memory Manager. If the `BUSY` flag is nonzero, the Memory Manager may be in the middle of a call, so it is not safe to call it.

### What routines must check the BUSY flag?

Classic desk accessory main routines and shutdown routines do not need to check the `BUSY` flag. If the Event Manager is active, the CDA gets control during `GetNextEvent`, not at interrupt time. If the Event Manager is not active, the CDA gets control only when the `BUSY` flag reaches zero.

GS/OS signal handlers do not need to check the BUSY flag, because the system dispatches signals only when the BUSY flag is zero.

Run Queue tasks do not need to check the `BUSY` flag before calling the Memory Manager. The system dispatches Run Queue tasks at `SystemTask` time—the `BUSY` flag may not be zero, but no Memory Manager call will be in progress.

Heartbeat interrupt tasks and other interrupt handlers **do** need to check the `BUSY` flag before calling the Memory Manager.

### Interrupt-time use of moveable memory blocks

If an interrupt-time routine needs access to an unlocked, non-fixed memory block, you must check the `BUSY` flag. It is not sufficient to lock the block, use it, and then unlock it (even if you twiddle the handle's access word directly). If the `BUSY` flag is non-zero, the Memory Manager could be in the middle of compacting memory, which means your block could be "in transit" from one address to another (some bytes copied, some not).

To use already-allocated memory at interrupt time, either keep the block locked or fixed, or check that the `BUSY` flag is zero before using the memory at interrupt time.

### What if BUSY is nonzero?

If the `BUSY` flag is nonzero, you may want to (depending on your application) exit the interrupt routine and hope the `BUSY` flag is zero the next time, or call `SchAddTask` in the Scheduler to make the system call your routine when the `BUSY` flag next returns to zero. Keep in mind, though, that only four scheduled tasks can be pending at a time.

## Interrupt-time flag byte

If the byte at location $E100CB is non-zero, the Memory Manager will not move any memory blocks, and it will not purge any blocks while trying to allocate memory (`PurgeHandle` and `PurgeAll` will still purge blocks).

Debugging utilities may temporarily increment this byte to allocate memory in situations when it is not safe for existing memory blocks to be moved or purged.

This flag byte is for use **only** by debugging aids and System Software. It would be mind-numbingly stupid for an application to use this flag instead of using `HLock` and `HUnlock`, since the advantages of a Memory Manager architecture with relocatable blocks would be lost.

It is not useful to check the value of the $E100CB flag. It is always set during interrupt handling whether any non-reentrant system component is busy or not.

Apple II
Technical Notes

Developer Technical Support

# Apple IIGS
# #58: Keyboard Modifiers Register Anomaly

Written by: Dave Lyons July 1989

This Technical Note discusses an anomaly with the keyboard modifiers register at location $C025 which prevents it from always properly reflecting the state of the Control and Shift keys.

---

There are two cases where pressing the Control key turns on the Shift bit instead of the Control bit in the keyboard modifiers register:

- An arrow key (or a Control key equivalent to an arrow key) is being held down and is repeating
- The Space bar or Delete key is being held down and repeating with the Fast Space/Delete option selected in the Control Panel

Since the Event Manager reads the modifiers byte, desktop applications may be affected by this anomaly.

## Further Reference

- *Apple IIGS Hardware Reference*



# Apple IIGS
# #59: Do Not Create Zero-Length Text Scraps

Revised by: Dave Lyons January 1991
Written by: Dave Lyons July 1989

This Technical Note described a problem with zero-length text scraps.
**Changed since July 1989:** This Note is obsolete beginning with System Software 5.0.3. There is no longer a problem with creating a text scrap of length zero.

---

In System Software 5.0.3 and later, `LEFromScrap` no longer trashes memory if you create a text scrap (scrap type 0) with length zero.

## Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2



# Apple IIGS
# #60: Menu Manager Memorabilia

Revised by: Matt Deatherage, Dave Lyons, & Tim Swihart November 1990
Written by: Dave Lyons July 1989

This Technical Note discusses the Menu Manager, specifically a few anomalies and some tips for making menus your friends.
**Changes since May 1990**: Noted that System Software 5.0.3 fixes a bug in `NewMenuBar2`.

## The Menu Manager Is Your Friend

In general, this is the truth. You can do all kinds of nifty things with menus, especially in System Software 5.0 and later. However, there are a few things you should know unless you generally are fond of pain in your life.

## Disabling Menus Gracefully

As documented, `SetMenuFlag` can be used to disable and enable entire menus. When a menu is disabled, the menu title and all items within the menu are disabled. You may pull down a disabled menu, but you may not select any item within it (unless the routine `MenuGlobal` has been used to allow inactive menu items to be selected).

Volume 1 of the *Apple IIGS Toolbox Reference* says you should call `DrawMenuBar` if you change the appearance of a menu title with `SetMenuFlag`. You can do this; this is fine. It may, however, induce dizziness if used often.

A more graceful way to dim menus is to follow `SetMenuFlag` with `HiliteMenu`. Calling `HiliteMenu` causes the menu title to be redrawn to reflect the current (or new) highlighting and menu flags. Using `HiliteMenu` instead of `DrawMenuBar` allows you to disable and enable menus gracefully, without noticeable flicker or threat of nasty patent infringement lawsuits from strobe light manufacturers.

## "System" Bars Versus "Window" Bars

As far as the Menu Manager is concerned, there are only two kinds of menu bars. One kind is in a window and the other kind is not. The former are called "window" menu bars and the latter are generally referred to as "system" menu bars.

Most people think of the System bar as the big menu bar across the top of the screen. This is encouraged by calls like `SetSysBar`, which takes a menu bar handle and sets the menu bar across the top of the screen to that menu bar. Trying to rename one or the other of these two concepts at this point is probably useless; instead, this Note refers to the bar across the top of the screen as the "System" bar (with a capital S), and menu bars not in windows as "system" bars (with a lowercase s).

When you start the Menu Manager, it creates an empty System bar for you. Before System Software 5.0, most people simply called `NewMenu` and `InsertMenu` to insert menus into that System bar. All was well in the world.

When 5.0 was released, it became very easy to create a new menu bar and all the menus within it using the `NewMenuBar2` call. This avoids a lot of code, and many new people use it. The problem comes with `DrawMenuBar`. If you simply call `NewMenuBar2` to obtain your menu bar and menus from resources, then call `DrawMenuBar` to make them visible, you usually get an empty menu bar. Why? The `windowPtr` parameter passed to `NewMenuBar2` determines whether or not the new menu bar created is a system bar or a window bar—it does **not** force the new bar to be the System (note the capital 'S') bar. So when `DrawMenuBar` draws the current System bar, it hasn't changed from the empty default one created by `MenuStartUp`.

This is why Volume 3 of *Apple IIGS Toolbox Reference* recommends code similar to the following:

```
menuHandle := NewMenuBar2(refDesc,menuBarTRef,NIL);
SetSysBar(menuHandle);
SetMenuBar(NIL);        {NIL makes the System bar the current menu bar}
```

if you want your menu bar to be the one across the top of the screen.

## A Bug in NewMenuBar2

`NewMenuBar2` is a handy thing to have around, but it does have a problem in 5.0.2 and earlier. When the Menu Manager is done with resources, it tries to use the internal toolbox call `CMReleaseResource` to free them in memory. However, it passes the wrong resource ID, and `CMReleaseResource` calls `SysFailMgr` if it encounters any errors at all (such as `Specified resource not found`).

What `NewMenuBar2` does improperly is push the high word of the resource ID onto the stack twice, instead of the high word followed by the low word. Because of the way the Resource Manager operates, `CMReleaseResource` returns with no error if the ID passed is `NIL`, but

the resource is not released (another good reason not to try to use the illegal value `NIL` as a resource ID).

If the high word of the menu bar resource is $0000, `NewMenuBar2` passes a resource ID of `NIL` to `CMReleaseResource`, which then doesn't quite release the resource, but returns no error. The menu bar resource hangs around in memory until `ResourceShutDown`. It's usually fairly small, so this is no loss. It still takes up less room than menu strings, which had to stay in memory until `MenuShutDown`.

If the high word of the menu bar resource is **not** zero, the bug causes `CMReleaseResource` to bring down the system. When using System Software 5.0.2 or earlier, make sure all menu bar resource IDs have a high word of $0000. System Software 5.0.3 fixes this bug.

## Menu and Menu Title ID Numbers

Table 13-4 in Volume 1 of *Apple IIGS Toolbox Reference* gives a listing of menu and menu item ID numbers. In both lists, $0000 and $FFFF are "reserved for internal use" and noted that $0000 usually indicates the first menu in the bar (or first item in the menu) and $FFFF usually indicates the last menu in the bar (or last item in the menu). Some developers have taken this to mean that they should give their first menu an ID of $0000 and their last one an ID of $FFFF.

This assumption is incorrect.. The Menu Manager may change these values internally to reflect such IDs, but they must not be assigned that way by an application. Some applications that use IDs of $0000 or $FFFF break under System Software 5.0 and later. Note that $0000 **can** be used as the `insertAfter` parameter to `InsertMenu` to insert a menu at the left of a menu bar, but $FFFF is not a valid `insertAfter` value.

## Desk Accessories and Menus

Some desk accessory developers would like to have their NDAs insert a menu in the System menu bar. While the menu itself can be inserted, the NDA **cannot** detect that a user has selected an item within that menu. The application gets the event and does not know what to do with it. NDAs that need a menu can put a menu bar in their own window. Since the `mouseDown` event then happens within the NDA's window, the NDA gets the event and can handle it normally. Be sure to make the NDA's menu bar the current menu bar before calling `MenuSelect` from within your NDA (to avoid possible conflicts between NDA menu item IDs and application menu item IDs). Restore the current menu bar to the application's menu bar before returning control to the application. Failure to do so prevents the application from finding its menus. Apple IIGS Technical Note #3, Window Information Bar Use documents how to put a menu in a window's information bar.

## Documentation Error in MenuSelect

Volume 1 of *Apple IIGS Toolbox Reference* states that `MenuSelect` returns the menu ID and the item ID of the selected item in the `when` field of the event record. This is incorrect. `MenuSelect` actually returns the information in the `wmTaskData` field of the **task record** (and this, in fact, is why you pass a task record and not just an event record to `MenuSelect`).

## Menu Strings and Bank Boundaries

`NewMenu` takes a pointer to a string; this string must **not** cross a bank boundary. If it does, a menu containing random garbage may result.

If your `NewMenu` strings are contained in your code segments, everything is fine—code segments cannot cross bank boundaries. Depending on your development environment, strings that are not in a code segment may or may not be allowed to cross bank boundaries. If you can find no other way to guarantee the strings do not cross a bank boundary, use `NewHandle` to allocate blocks with attributes $4010 (fixed, no bank cross) and copy the strings to these blocks.

If you create menus from resources, be sure the resources have their `noCrossBank` attribute bits set. Note that a memory block that **can** cross a bank boundary usually does **not**, so your application may be working by accident.

Note that this restriction applies only to menu strings, not the menu templates that can be used with `NewMenu2`.

## Return Values From GetMenuTitle and GetMIItem

Starting with System Software 5.0, `GetMenuTitle` and `GetMItem` can return handles and resource IDs, not just pointers. The type of data returned depends on how the menu or item was created, so existing applications are not affected. For more information, see *Apple IIGS Toolbox Reference*, Volume 3, Chapter 37, "New Features of the Menu Manager."

### Further Reference

- *Apple IIGS Toolbox Reference,* Volumes 1 & 3
- Apple IIGS Technical Note #3, Window Information Bar Use



# Apple IIGS
# #61: Window Title Handles

Written by: Dave Lyons July 1989

This Technical Note discusses extensions to `SetWTitle` and `GetWTitle` in System Software 5.0 and later which allow handles to be used as window titles.

---

Prior to System Software 5.0, window titles were pointers to Pascal-style strings (with a leading length byte), but now window titles can be stored in handles, with bit 31 of `titlePtr` set to indicate that the parameter is actually a handle.

Once you call `SetWTitle` with a handle for the title parameter, the handle belongs to the Window Manager, which will dispose of it when the window is closed or retitled. You must not dispose of the handle yourself, and you must not change the data it contains.

## Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2



# Apple IIGS
# #62: No Non-Solid Window Background Patterns

Written by: Dave Lyons July 1989

This Technical Note discusses why window background patterns should always be solid; non-solid patterns are not always drawn with the expected alignment.

---

When the Window Manager erases part of a window's content area to its port's background pattern, it is not always aligned with already-drawn parts of the window. With a solid background pattern, this has no visible effect; however, if you try to use a grid, for example, the effect is obvious.

To simulate a non-solid background pattern, just erase the desired area to the pattern you want in your update routine. For best results, use a solid background pattern of the color most common in the pattern you really want.

For example, if you want a white grid on a black background, give the window a solid black background pattern, and use `FillRect` during the update routine to draw the grid. If you keep the default white background pattern, the end result will be the same, but your window content will briefly be solid white before your update routine fills it with your pattern.

## Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2



# Apple IIGS
# #63: Master Color Values

Revised by: Dave Lyons May 1991
Written by: Jim Luther July 1989

This Technical Note documents master color values used for the Apple IIGS text, text background, and border colors.

**Changes since July 1989:** Added information on the standard QuickDraw II 640-mode color table and provided a 320-mode color table that produces similar colors.

## Border Color Values

There are times when you may want to make parts of the IIGS Super Hi-Res screen the same color as the text, text background, and border colors. This is particularly useful when using the Apple II Video Overlay Card. Table 1 lists each color using the names from the Control Panel CDA, the color register values used for that color by the color registers, and the master color value used for that color by the Super Hi-Res screen.

| Color Name | Color Register Value | Master Color Value |
|---|---|---|
| Black | $0 | $0000 |
| Deep Red | $1 | $0D03 |
| Dark Blue | $2 | $0009 |
| Purple | $3 | $0D2D |
| Dark Green | $4 | $0072 |
| Dark Gray | $5 | $0555 |
| Medium Blue | $6 | $022F |
| Light Blue | $7 | $06AF |
| Brown | $8 | $0850 |
| Orange | $9 | $0F60 |
| Light Gray | $A | $0AAA |
| Pink | $B | $0F98 |
| Light Green | $C | $01D0 |
| Yellow | $D | $0FF0 |
| Aquamarine | $E | $04F9 |
| White | $F | $0FFF |

**Table 1–Master Color Values**

The *Apple IIGS Hardware Reference* documents the color registers at $C022 and $C034, and the *Apple IIGS Toolbox Reference*, Volume 2 documents the master color values.

## Standard 640-mode Color Table

The description of dithering on pages 16-35 and 16-36 of Apple IIgs Toolbox Reference, Volume 2 is correct, but some of the color values in Table 16-5 are incorrect. Table 2 lists the standard QuickDraw II 640-mode color table:

| Color Table Offset | Color Name | Master Color Value |
|---|---|---|
| $0 | Black | $0000 |
| $1 | Red | $0F00 |
| $2 | Green | $00F0 |
| $3 | White | $0FFF |
| $4 | Black | $0000 |
| $5 | Blue | $000F |
| $6 | Yellow-green | $0FF0 |
| $7 | White | $0FFF |
| $8 | Black | $0000 |
| $9 | Red | $0F00 |
| $A | Green | $00F0 |
| $B | White | $0FFF |
| $C | Black | $0000 |
| $D | Blue | $000F |
| $E | Yellow-green | $0FF0 |
| $F | White | $0FFF |

**Table 2–Standard 640-mode Color Table**

Table 3 shows Master Color values you can use in 320-mode to get close approximations of the sixteen standard 640-mode "solid" (really dithered) 640-mode colors.

| Color Table Offset | Color Name | Master Color Value |
|---|---|---|
| $0 | Black | $0000 |
| $1 | Deep Blue | $0008 |
| $2 | Yellow-brown | $0880 |
| $3 | Gray | $0888 |
| $4 | Red | $0800 |
| $5 | Purple | $0808 |
| $6 | Orange | $0F80 |
| $7 | Pink | $0F88 |
| $8 | Dark Green | $0080 |
| $9 | Aqua | $0088 |
| $A | Bright Green | $08F0 |
| $B | Pale Green | $08F8 |
| $C | Gray | $0888 |
| $D | Periwinkle Blue | $088F |
| $E | Yellow | $0FF8 |
| $F | White | $0FFF |

**Table 3–Standard 640-mode Color Table**

## Further Reference

- *Apple IIGS Hardware Reference*, pp. 58, 76-78
- *Apple IIGS Toolbox Reference*, Volume 2, p. 16-31



# Apple IIGS
# #64: Apple IIGS Installer and Installer Scripts

Revised by: Jim Luther September 1989
Written by: Jim Luther & "Jay" Schaffer July 1989

This Technical Note describes how the Apple IIGS Installer program executes script files and documents how to write script files for it. Note that some of the information in this Note is specific to Installer V1.10.

**Changes since July 1989:** Changed the `sourcePrefix` and `sourcePathname` field descriptions, since `sourcePrefix` must not be empty if any `sourcePathname` fields are partial pathnames.

## Introduction

The Apple IIGS Installer, a utility program that is included with Apple IIGS System Software, can be used to install System Software or applications on a given volume. "Scripts" control the Installer, and they are simply lists of files with information about where and how to install those files. The user interface of the Installer is described in the *Apple IIGS System Tools Manual*. This Note describes how the Installer executes scripts and how to write scripts to install your applications.

## Installer Setup on Disk

Setting up the Installer on your application disk is a simple procedure.

1. Copy the Installer program to your application disk.
2. Create a subdirectory (folder) named Scripts at the same directory level as the Installer program.
3. Copy your scripts into the Scripts subdirectory.

## How the Installer Processes Scripts

The Installer reads script files into memory in their entirety, parses them, strips them of all comments, compacts them, then verifies them. It then checks the `scriptFlags` field to see if a Caution alert should be displayed. This facility permits the script writer to force the user to

read the script's help message and make a choice to either continue with file manipulations or skip the installation altogether, which is especially useful when a script installation would be inappropriate on a certain volume.

The Installer then executes the script in two passes. The first pass determines if the update can be completed by calculating the total size of the files to be deleted from the destination volume and of the files to be installed. If there is not sufficient room on the destination volume, the Installer determines the amount of additional space required to complete installation (number of blocks needed divided by two, plus one), reports this result to the user in terms of kilobytes, then terminates execution of the script. It is impossible to determine directory block requirements with complete accuracy. The Installer's space calculation algorithms are good, but they are not perfect.

If the first pass determines that there is sufficient room for the complete update, the Installer continues with the second pass, deleting and copying files in accordance with the instructions contained in the script flags. The Installer "blindly" unlocks locked files and folders, creates necessary subdirectories if they do not already exist, and replaces requested files without regard to version numbers or creation dates of existing files.

The user may terminate execution of any script (and of those which follow) by pressing the Open-Apple–Period (-.) key combination. The Installer checks for key-down events between every file transfer and at the end of the first pass. If the user requests termination, the Installer warns of the possibility of leaving an unknown mix of file versions on the volume and gives the user the opportunity to continue with the installation or to terminate as requested. (See the "Error Handling" section for more details.)

Scripts are typically written with the ability to remove all of their related files from a particular volume (i.e., in case of an accidental installation); however, they do not have the ability to remove directories which contain files (even if the script installed them), and they can neither recover nor list files which were deleted during the installation process.

After processing all the instructions in a script, the Installer checks to see if additional scripts are selected, and, if they are, it executes them in the order in which they appear in the update selection window until all scripts are successfully completed. Once all selected scripts are completed, the Installer notifies the user that the installation or removal process was successful.

It is important to note several facts about script execution:

- Each script is processed from beginning to end as if it were the only script selected.
- If the execution of a script generates an error, or if the user terminates further processing of a script, the queue is cleared of any additional scripts waiting to be executed and control returns to the user.
- It is possible for the Installer to execute several scripts successfully before encountering one which cannot be executed due to insufficient space on the destination volume.
- All selected scripts use the folder that the user selects as the "Application Folder."

If a user installs or removes system files (i.e., tools, fonts, drivers, etc.) from the boot volume, it may create problems. Therefore, whenever a system level update occurs on a boot volume, the Installer disables all desk accessories and closes the Sys.Resources file. When the user quits the Installer after a system level update, it alerts the user of the need to restart the system, and the default response to this alert is to restart.

## Error Handling

### User Cancel Request

If the user cancels script execution any time after it has started (i.e., by pressing the Open-Apple–Period (-.) key combination), the Installer treats it as an error condition since there is likely an unknown mix of file versions on the volume. In this case, the Installer gives the user the opportunity to continue with the installation or to terminate as requested. A user-initiated cancel request is not acknowledged until the current file copy or delete request is complete. Terminating script execution also clears the queue of other scripts waiting for execution and returns control to the user.

### Non-Recoverable Errors

Some errors are simply fatal. If a directory or file is corrupted, the media is bad, or the selected script is longer than 65,535 bytes, the Installer halts execution of the script and alerts the user that a fatal error has occurred with a Stop alert box. Clicking the OK button in this alert box clears the queue of other scripts waiting for execution and returns control to the user.

### Script Errors and File Not Found Errors

When the Installer detects a script error or a File Not Found error, it reports the name of the source file and destination file it was processing with the normal error message. This additional information should help script writers find the offending `fileSpecification` field. If the error is associated with the header, no filename is reported. This condition clears the queue of other scripts waiting for execution and returns control to the user.

### Volume Not Found Errors

Volume Not Found errors produce a dialog box prompting the user to insert the missing volume. If the user clicks the OK button, the Installer attempts the file access call again, but if the user clicks the Cancel button, the Installer flags it as an error condition, clears the queue of other scripts waiting for execution, and returns control to the user.

## Script File Composition

A script is simply a list of instructions for the Installer, and it can specify that files be copied from a source volume to a destination volume (or directory, when applicable) or that files be

removed from a destination volume. Script files are ASCII files (file type $04) containing printable ASCII characters (i.e., with the high-bit clear). The directory in which the Installer resides must contain a directory named Scripts, in which all script files visible to that copy of the Installer must be located. Script files may not exceed 65,535 bytes in length. Any attempt to execute a script larger than this size produces a non-recoverable error.

A script consists of a `header` field followed by any number of `fileSpecification` and `comment` fields. These fields are separated by tildes (~). Two consecutive tildes signal the end of the script, and any additional characters past the end of script marker are ignored. Figure 1 shows the syntax diagram for a script.

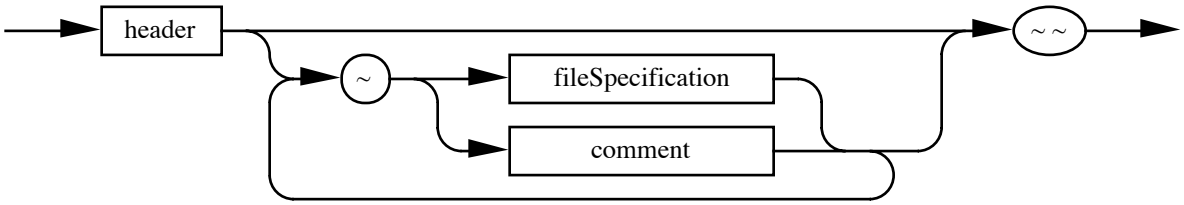


**Figure 1–Script Syntax Diagram**

### header Field

The `header` field consists of the `scriptIdentifier`, `scriptVersion`, `scriptFlag`, `scriptName`, and `scriptHelp` fields, and it may also contain an optional `sourcePrefix` field. These fields supply the installer with general information about the script file. No comments are permitted within the `header` field. Figure 2 shows the syntax diagram for the `header` field.

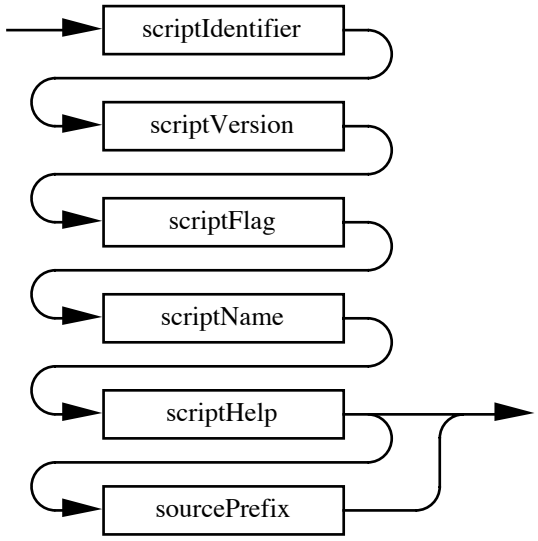


**Figure 2–header Field Syntax Diagram**

The `scriptIdentifier` field identifies the text file as a script file, and it consists of eight characters (“SCRIPT” followed by two carriage returns, or 53 43 52 49 50 54 0D 0D in hexadecimal). Figure 3 shows the syntax diagram for the `scriptIdentifier` field.

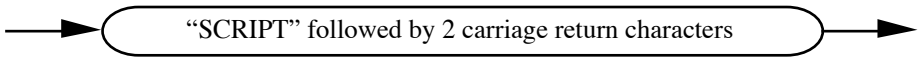


**Figure 3–scriptIdentifier Field Syntax Diagram**

The `scriptVersion` field defines the minimum version of the Installer program that can read and execute the instructions in this script file. It should normally consist of seven characters (“V1.10” followed by two carriage returns, or 56 31 2E 31 30 0D 0D in hexadecimal).

Version 1.0 of the Installer moves **only** the data fork and does not return an error. For compatibility with the original release of the Installer, the value of `scriptVersion` is V1.00. Scripts which move extended files (i.e., files with resource forks) or work with an AppleShare volume **must** have a `scriptVersion` of V1.10. Figure 4 shows the syntax diagram for the `scriptVersion` field.

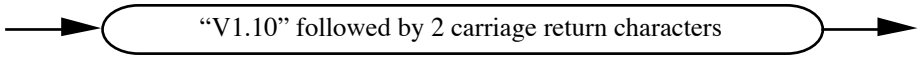

**Figure 4–scriptVersion Field Syntax Diagram**

The `scriptFlag` field defines the directory requirements of the script file. The first character of the `scriptFlag` field must be either the uppercase character "R" (indicating that the installation must occur at the root directory, such as in a System Software update) or the uppercase character "X" (indicating that the user must specify the directory where installation should take place).

The second character of the `scriptFlag` field must be either an uppercase or lowercase character "R" (indicating that the Remove command is valid for this script) or an uppercase or lowercase character "N" (indicating that the Remove command is not valid and the button should be dimmed and inactive). If this character is lowercase, before any file manipulations begin, the Installer displays a Caution alert with the contents of the `scriptHelp` field and button controls to permit the user to choose whether to execute the script or to skip it and go to the next script, if any.

For example, a `scriptFlag` field might contain the following four characters: "Rr" followed by two carriage returns, or 52 52 0D 0D in hexadecimal. Figure 5 shows the syntax diagram for the `scriptFlag` field.

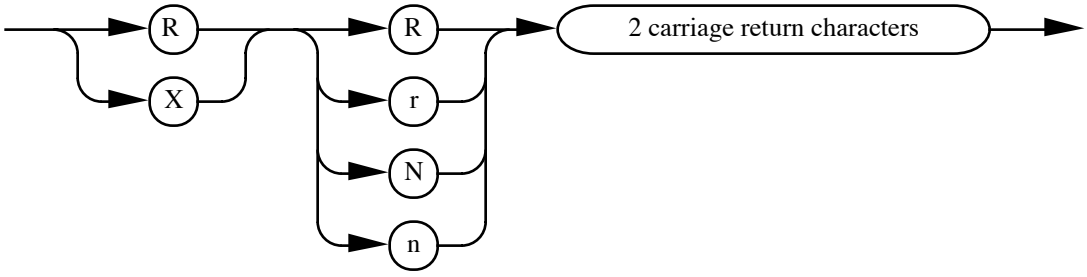


**Figure 5–scriptFlag Field Syntax Diagram**

The `scriptName` field defines the name of the script as it appears in the Installer's script selection window. It is recommended that care be taken to use a name that fits within the display window. This field consists of any number of characters ending with a carriage return and may not include a tilde or carriage return. An example of `scriptName` might be: "Example Script" followed by a carriage return, or 45 78 61 6D 70 6C 65 20 53 63 72 69 70 74 0D in hexadecimal. Figure 6 shows the syntax diagram for the `scriptName` field.

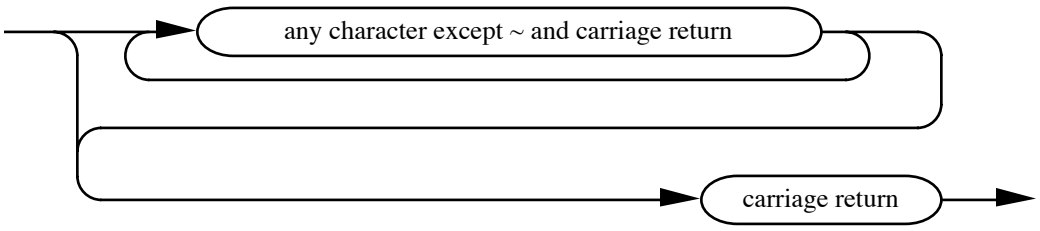


**Figure 6–scriptName Field Syntax Diagram**

The `scriptHelp` field defines the text which appears when the user clicks the Help button. It is recommended that care be taken to ensure the text fits within the help window. This field consists of any number of characters ending with two backslashes (\\) and a carriage return. It may not include two consecutive backslashes or a tilde; however, it may include carriage returns. An example of `scriptHelp` might be: "Help\\" followed by a carriage return, or 48 65 6C 70 5C 5C 0D in hexadecimal. Figure 7 shows the syntax diagram for the `scriptHelp` field.

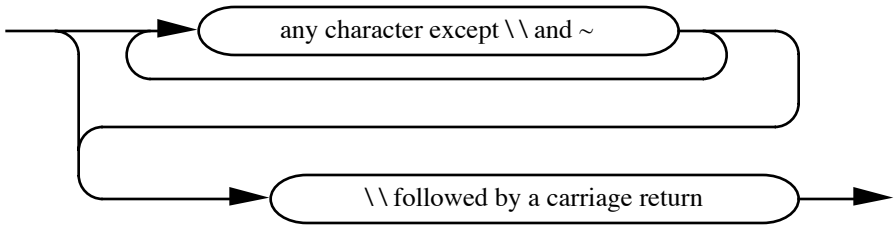


**Figure 7–scriptHelp Field Syntax Diagram**

The optional `sourcePrefix` field is the prefix used with source files defined by partial pathnames. Either slashes (/) or colons (:) may be used as the pathname separator character. If there is no `sourcePrefix`, this entry must be empty. If no `sourcePrefix` is specified, all `sourcePathname` fields used within `fileSpecification` fields must be full pathnames. An example of a `sourcePrefix` might be: ":System.Disk:System", or 3A 53 79 73 74 65 6D 2E 44 69 73 6B 3A 53 79 73 74 65 6D in hexadecimal. Figure 8 shows the syntax diagram for the `sourcePrefix` field. *GS/OS Reference* defines legal pathnames and prefixes.


**Figure 8–sourcePrefix Field Syntax Diagram**

### fileSpecification Field

A `fileSpecification` field contains the instructions to copy a file to or remove a file from the destination volume (or directory, when applicable). A `fileSpecification` field is composed of the `fileSpecWorkspace`, `fileSpecFlags`, `fileTypeAuxType`, `createDateTime`, `sourcePathname`, and `destinationPathname` fields. The script may contain as many `fileSpecification` fields as necessary. Figure 9 shows the syntax diagram for the `fileSpecification` field.

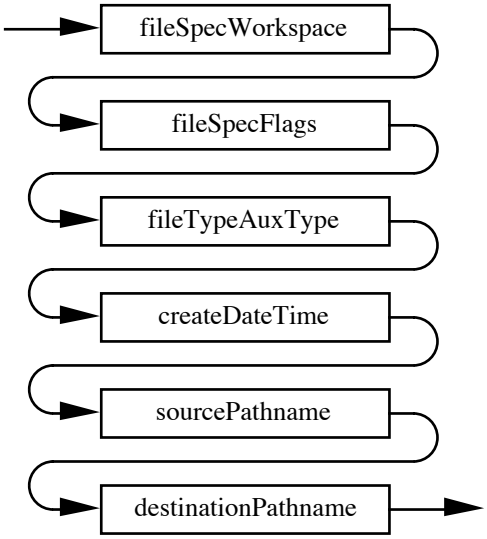


**Figure 9–fileSpecification Field Syntax Diagram**

The `fileSpecWorkspace` field is 16 bytes that the Installer uses for work space, it can contain any character except a tilde, and it may not begin with a tilde or an asterisk (*). It is suggested that 15 readable characters followed by a carriage return might be easiest to see and count. An example of `fileSpecWorkspace` might be: ":::Workspace:::" followed by a carriage return, or 3A 3A 3A 57 6F 72 6B 73 70 61 63 65 3A 3A 3A 0D in hexadecimal. Figure 10 shows the syntax diagram for the `fileSpecWorkspace` field.

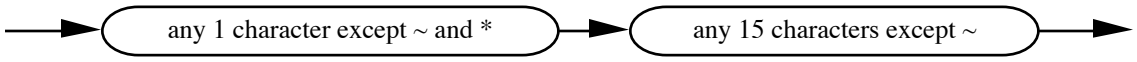


**Figure 10–fileSpecWorkspace Field Syntax Diagram**

The `fileSpecFlags` tell the Installer what this `fileSpecification` does. The `fileSpecFlags` field consists of the `requiredFlag` field followed by the `optionalFlags` field and a carriage return. Figure 11 shows the syntax diagram for the `fileSpecFlags` field.


**Figure 11–fileSpecFlags Field Syntax Diagram**

The `requiredFlag` field tells the Installer what to do with this `fileSpecification` when the Install or Remove buttons are used. The `requiredFlag` field must start with only one of the following characters: 1, 2, 3, or 4, and it must end with a carriage return. Any number of characters (except tilde and carriage return ) may fall between the flag character and the ending carriage return. These additional characters are ignored by the Installer, making it possible to place comments within a `requiredFlag` field. Figure 12 shows the syntax diagram for the `requiredFlag` field.

The four `requiredFlag` characters tell the installer the following:

1 If the user clicks the Install button, delete the `destinationPathname` from the destination volume, if it exists, and copy the file from the source volume. If the user clicks the Remove button, delete the `destinationPathname` from the destination volume, if it exists.
2 If the user clicks the Install button, delete the `destinationPathname` from the destination volume, if it exists, and copy the file from the source volume. If the user clicks the Remove button, do nothing.
3 If the user clicks the Install button, delete the `destinationPathname` from the destination volume, if it exists. If the user clicks the Remove button, delete the `destinationPathname` from the destination volume, if it exists.
4 If the user clicks the Install button, delete the `destinationPathname` from the destination volume, if it exists. If the user clicks the Remove button, do nothing.

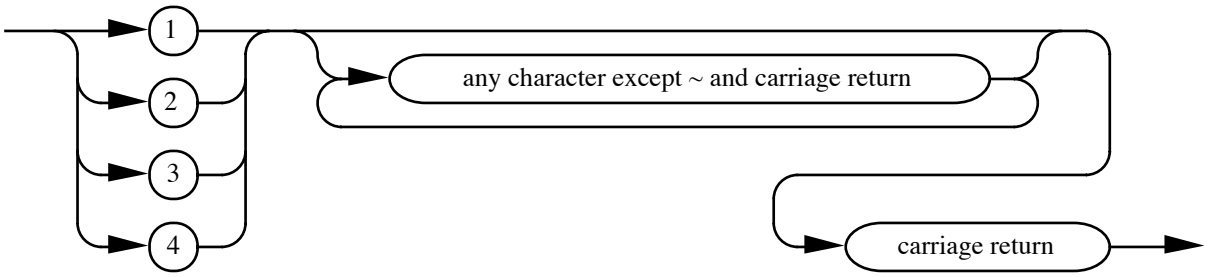


**Figure 12–requiredFlag Field Syntax Diagram**

The `optionalFlags` field gives the Installer additional duties to perform with this `fileSpecification` when the Install or Remove buttons are used. The five option fields, B, C, D, F, and U (must be uppercase), within the `optionalFlags` field are formatted the same as the `requiredFlag` field. Figure 13 shows the syntax diagram for the `optionalFlags` field.

The five `optionalFlags` characters tell the installer the following:

B This flag instructs the Installer to replace the boot code on blocks zero and one of the destination volume. The boot code replacement `fileSpecification` is reserved for use by Apple Computer, Inc.

C The creation date and time of the file designated by the `sourcePathname` field must match the `createDateTime` entry in this `fileSpecification` field.

D The designated `destinationPathname` should be deleted if, and only if, it has a creation date and time that is older than `createDateTime`. This flag must be used with a "4" `requiredFlag`.

F The file type and auxiliary type of the file designated by the `sourcePathname` must match the `fileTypeAuxType` field in this `fileSpecification` field.

U Update (replace) the existing `destinationPathname` only if it exists. This flag must be used with a "1" or a "2" `requiredFlag`.

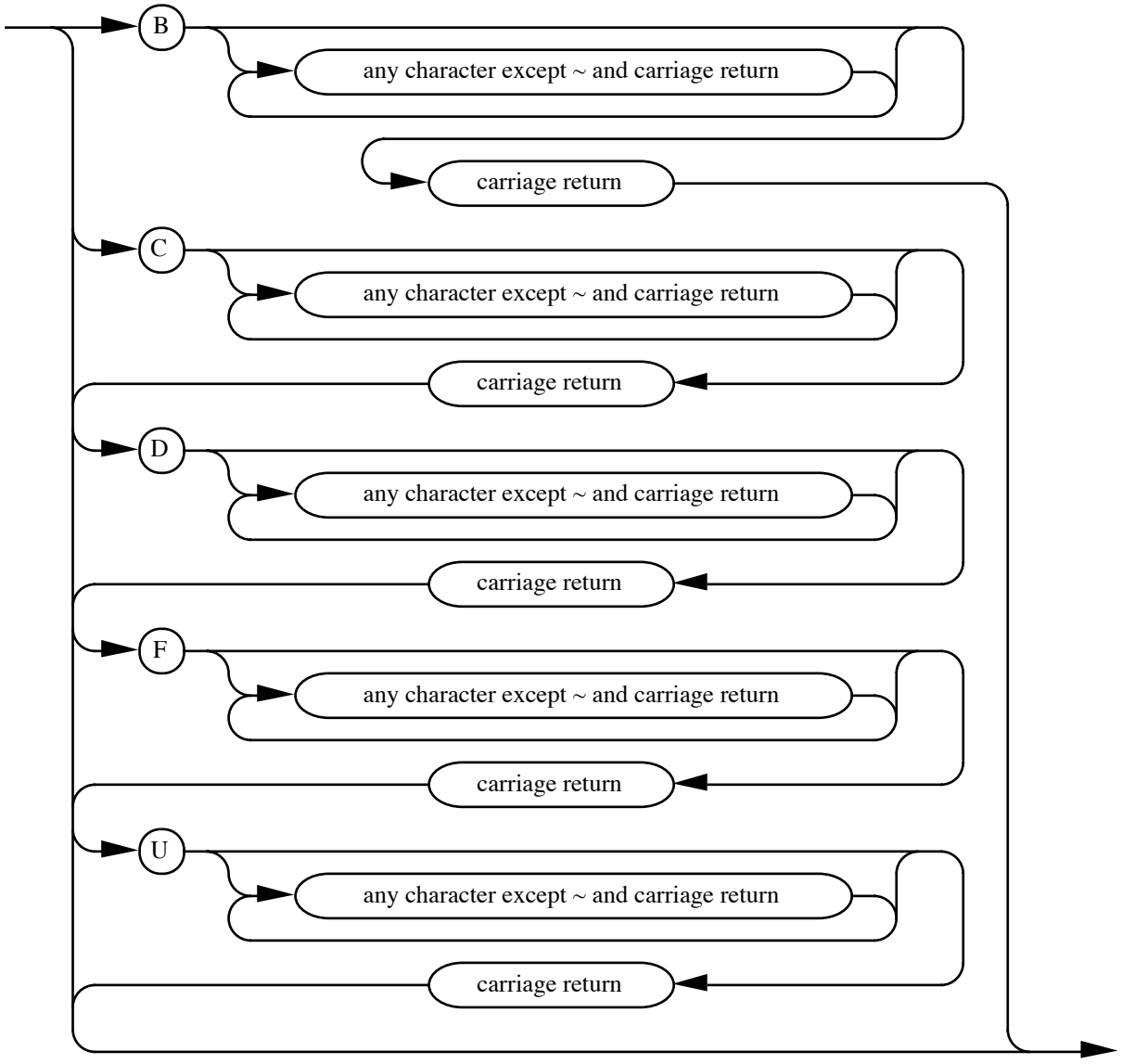


**Figure 13–optionalFlags Field Syntax Diagram**

The `fileTypeAuxType` field is used if the "F" `optionalFlags` field is present in the `fileSpecification` field. If the `fileTypeAuxType` field is used, it must start with a `fileType` field and an `auxType` field and must end with a carriage return. Any number of characters (except tilde and carriage return) may fall between the `auxType` field and the ending carriage return. These additional characters are ignored by the Installer, making it possible to place comments within the `fileTypeAuxType` field. If the "F" `optionalFlags` field is not used, then the fileTypeAuxType field must be only a carriage return. For a list of current file types and auxiliary types, see the Apple II File Type Notes. Figure 14 shows the syntax diagram for the `fileTypeAuxType` field.

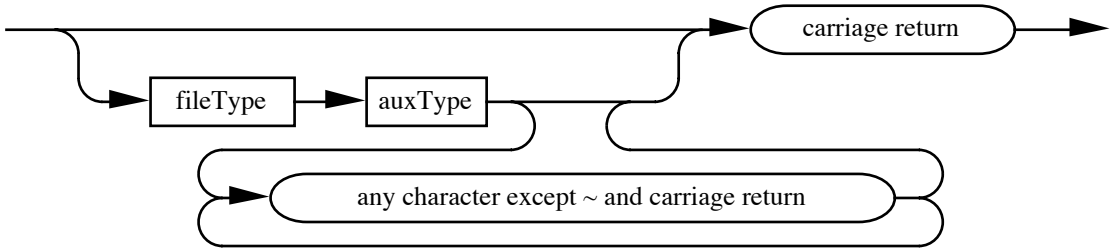


**Figure 14–fileTypeAuxType Field Syntax Diagram**

The `fileType` part of the `fileTypeAuxType` field consists of four, and only four, hexadecimal digits. These four digits identify a GS/OS file type if the "F" `optionalFlags` field is present in the `fileSpecification` field. An example of `fileType` might be: "00B3", or 30 30 42 33 in hexadecimal. Figure 15 shows the syntax diagram for the `fileType` field.

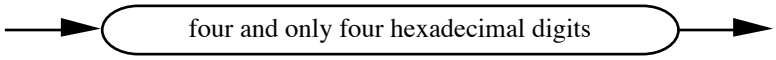


**Figure 15–fileType Field Syntax Diagram**

The `auxType` part of the `fileTypeAuxType` field consists of eight, and only eight, hexadecimal digits. These eight hexadecimal digits identify a GS/OS auxiliary type if the "F" `optionalFlags` field is present in the `fileSpecification` field. An example of `auxType` might be: "00000000", or 30 30 30 30 30 30 30 30 in hexadecimal. Figure 16 shows the syntax diagram for the `auxType` field.

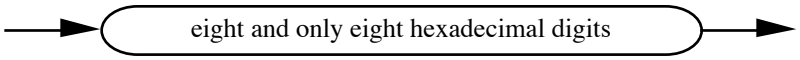


**Figure 16–auxType Field Syntax Diagram**

The `createDateTime` field is used if the “C” or “D” `optionalFlags` fields are present in the `fileSpecification` field. If the `createDateTime` field is used, it must start with a `date` field, a single space and a `time` field and must end with a carriage return. Any number of characters (except tilde and carriage return) may fall between the `time` field and the ending carriage return. These additional characters are ignored by the Installer, making it possible to place comments within the `createDateTime` field. If the “C” or “D” `optionalFlags` fields are not used, then the `createDateTime` field must be only a carriage return. Figure 17 shows the syntax diagram for the `createDateTime` field.

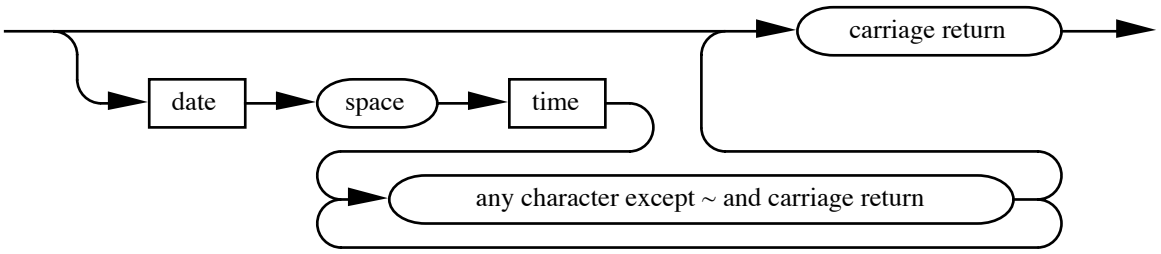


**Figure 17–createDateTime Field Syntax Diagram**

The `date` subfield of the `createDateTime` field is nine ASCII characters consisting of the day of the month, a space, a three-character month abbreviation, a space, and the year. The day of the month is a two-character number between 01 and 31. The month abbreviation may be “Jan”, “Feb”, “Mar”, “Apr”, “May”, “Jun”, “Jul”, “Aug”, “Sep”, “Oct”, “Nov”, or “Dec” in any combination of uppercase and lowercase characters. The year is a two-character number between 00 and 99. An example of the `date` subfield might be: “31 Mar 57”, or 33 31 20 4D 61 72 20 35 37 in hexadecimal. Figure 18 shows the syntax diagram for the `date` subfield.

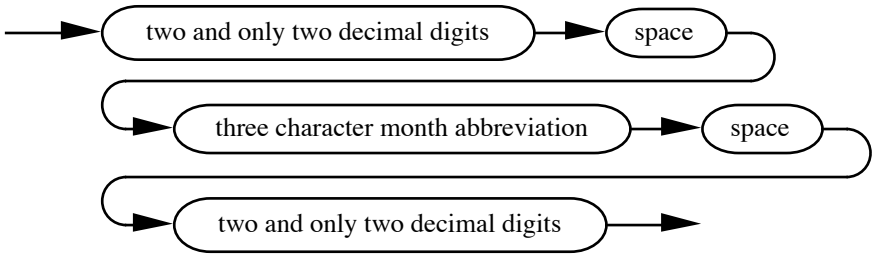


**Figure 18–date Field Syntax Diagram**

The `time` subfield of the `createDateTime` field is five ASCII characters consisting of the military format hour of the day, a colon, and the minute of the hour. The hour of the day is a two-character number between 00 and 23. The minute of the hour is a two-character number between 00 and 59. An example of the `time` subfield might be: “08:30”, or 30 38 3A 33 30 in hexadecimal. Figure 19 shows the syntax diagram for the `time` subfield.

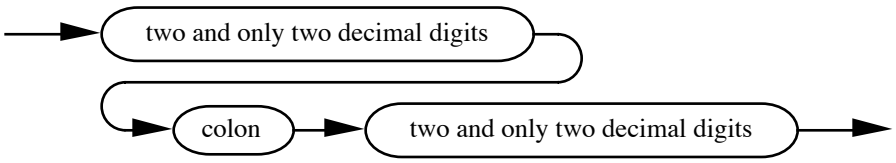

**Figure 19–time Field Syntax Diagram**

The `sourcePathname` field describes the name and location of the source file. The `sourcePathname` field consists of a valid GS/OS pathname followed by a carriage return. If the `sourcePathname` is a partial pathname, the `sourcePrefix` in the `header` field is used to complete the full pathname. If no `sourcePrefix` is specified in the `header` field, all `sourcePathname` fields must be full pathnames. If the `fileSpecFlags` indicate removal only, then the `sourcePathname` is a carriage return only. No optional comments are permitted in this field. Figure 20 shows the syntax diagram for the `sourcePathname` field. *GS/OS Reference* defines legal pathnames and prefixes.

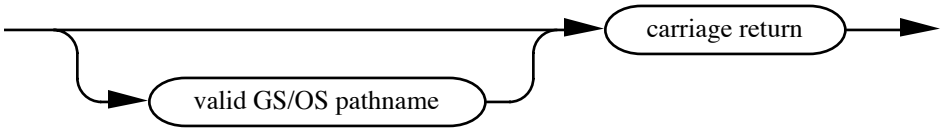


**Figure 20–sourcePathname Field Syntax Diagram**

The `destinationPathname` field describes the name and location of the destination file. The `destinationPathname` field consists of a valid GS/OS partial pathname (the prefix has already been set by the Installer to the location of the destination directory, either the root directory or a user selected directory) followed by a carriage return. No optional comments are permitted in this field. Figure 21 shows the syntax diagram for the `destinationPathname` field. *GS/OS Reference* defines legal pathnames and prefixes.

Note that GS/OS now allows filenames to contain both uppercase and lowercase characters. Although filenames are not case sensitive, you should be consistent in your use of uppercase and lowercase usage in the `destinationPathname` field. Whatever you use here is what everyone sees.

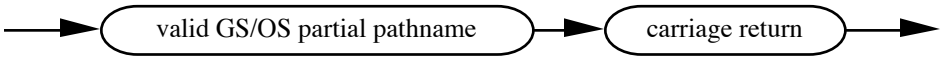


**Figure 21–destinationPathname Field Syntax Diagram**

**comment Field**

The `comment` field allows commenting script files. A `comment` field must begin with an asterisk. The Installer ignores all characters within a `comment` field, except tilde, and the `comment` field ends at the first tilde encountered. Figure 22 shows the syntax diagram for the `comment` field.

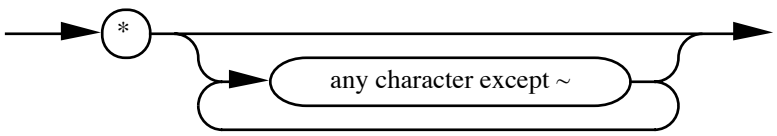


**Figure 22–comment Field Syntax Diagram**

## Examples

Now that the script language is described, it's time to look at a couple of example scripts. The first example, CD-ROM from the System.Tools disk, installs the files necessary for you to use CD-ROM drives. The CD-ROM script is an example of using the Installer to install or update existing software. The second example, Advanced Disk Utility from the System.Tools disk, installs the files necessary to update the Advanced Disk Utility program. The Advanced Disk Utility script is an example of using the Installer to install an application in any directory on the destination volume. In both examples (Examples 1 and 2), carriage returns are shown with a paragraph mark (¶) since they are used as delimiters within scripts.

### The CD-ROM Script

The `header` field starts with "SCRIPT" to identify this text file as a script file. The `scriptVersion` is "V1.10" because this script may have to copy the resource fork of a file. The `scriptFlag` field is "RR", which tells the Installer to install at the root directory level and that the Remove button is valid for this script. The second "R" character in the `scriptFlag` field is uppercase, which tells the Installer **not** to display a Caution alert with the contents of the `scriptHelp` field. The `scriptName` field is "CD-ROM". The `scriptName` is shown in the Installer's list of scripts. The `scriptHelp` field (everything between the `scriptName` field and the "\\" delimiter) is the text that will be displayed if the Installer's Help button is used. The `sourcePrefix` is ":SYSTEM.TOOLS". That is the name of the volume where the source files for this update are found.

After the `header` field, there is a single `comment` field and then five `fileSpecification` fields. The `comment` field starts at the asterisk after the first tilde and ends at the next tilde. All five `fileSpecification` fields start with the suggested 16-byte `fileSpecWorkSpace` (":::WorkSpace:::¶") and end at the next tilde.

The first, fourth, and fifth `fileSpecification` fields use the "1" `requiredFlag`. This flag tells the Installer to copy the `sourcePathname` to the `destinationPathname` if the Install button is used, or to delete the `destinationPathname` if the Remove button is used. Notice the three blank lines after the "1" `requiredFlag`. The first blank line marks the end of the `fileSpecFlags`. The `fileTypeAuxType` field, the second blank line, is blank because the "F" `optionalFlags` field is not used. The `createDateTime` field, the third blank line, is blank because the "C" and "D" `optionalFlags` are not used.

The second `fileSpecification` field uses the "3" `requiredFlag` to tell the Installer to delete the `destinationPathname`, "System:Drivers:SCSI.Driver", if either the Install or the Delete button is used. SCSI.Driver is the interim SCSI driver from System Software 4.0. The `sourcePathname` field, the fourth blank line after the "3" `requiredFlag`, is not needed since the "3" `requiredFlag` is used.

The third `fileSpecification` field uses the "2" `requiredFlag` to tell the Installer to delete the `destinationPathname`, "System:Drivers:SCSI.Manager" if the Install button is used. The Installer does **not** delete the `destinationPathname` if the Remove button is

used. The "2" `requiredFlag` prevents this script from removing SCSI.Manager, which might have been installed by another script.

Two consecutive tildes after the fifth `fileSpecification` field signal the end of this script.

```
SCRIPT¶
¶
V1.10¶
¶
RR¶
¶
CD-ROM¶
This script installs the files necessary for you to use CD-ROM drives.  The selected
disk must be a startup disk.\\¶
:SYSTEM.TOOLS~*¶
 This is the Installer script necessary to move the CD-ROM files from :SYSTEM.TOOLS to
the user's startup disk.¶
~:::Workspace:::¶
1¶
¶
¶
¶
System:FSTs:HS.FST¶
System:FSTs:HS.FST¶
~:::Workspace:::¶
3¶
¶
¶
¶
¶
System:Drivers:SCSI.Driver¶
~:::Workspace:::¶
2¶
¶
¶
¶
System:Drivers:SCSI.Manager¶
System:Drivers:SCSI.Manager¶
~:::Workspace:::¶
1¶
¶
¶
¶
System:Drivers:SCSICD.Driver¶
System:Drivers:SCSICD.Driver¶
~:::Workspace:::¶
1¶
¶
¶
¶
System:Desk.Accs:CDRemote¶
System:Desk.Accs:CDRemote¶
~~
```

**Example 1–CD-ROM Script**

### The Advanced Disk Utility Script

The `header` field starts with "SCRIPT" to identify this text file as a script file. The `scriptVersion` is "V1.10" because this script may have to copy the resource fork of a file. The `scriptFlag` field is "XR", which tells the Installer the user must specify the directory where the installation should take place and that the Remove button is valid for this script. The second character (R) in the `scriptFlag` field is uppercase, which tells the Installer **not** to display a Caution alert with the contents of the `scriptHelp` field. The `scriptName` field is "Advanced Disk Utility". The `scriptName` will be shown in the Installer's list of scripts. The `scriptHelp` field (everything between the `scriptName` field and the "\\" delimiter) is the text that will be displayed if the Installer's Help button is used. The `sourcePrefix` is ":SYSTEM.TOOLS". That is the name of the volume where the source files for this update are found.

After the `header` field, there is a single `comment` field then one `fileSpecification` field. The `comment` field starts at the asterisk after the first tilde and ends at the next tilde. The `fileSpecification` field starts with the suggested 16-byte `fileSpecWorkSpace` (":::WorkSpace:::¶") and ends at the next tilde.

The `fileSpecification` field uses the "1" `requiredFlag`. This tells the Installer to copy the `sourcePathname` to the `destinationPathname` if the Install button is used or to delete the `destinationPathname` if the Remove button is used.

Two consecutive tildes signal the end of this script.

```
SCRIPT¶
¶
V1.10¶
¶
XR¶
¶
Advanced Disk Utility¶
This script installs the files necessary to update the Advanced Disk Utility program.
These files will be installed on the selected disk.\\¶
:SYSTEM.TOOLS~*¶
 This is the Installer script necessary to update the Advanced Disk Utility file from
:SYSTEM.TOOLS to the user's disk.¶
~:::Workspace:::¶
1¶
¶
¶
¶
Adv.Disk.Util¶
Adv.Disk.Util¶
~~
```

**Example 2–Advanced Disk Utility Script**

### Further Reference

- *Apple IIGS System Tools Manual*
- *GS/OS Reference*