Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#80: QuickDraw II Clipping

Written by: Eric Soldan March 1990

This Technical Note explains a lot about QuickDraw II operation, specifically clipping.

Before Beginning

Before beginning this Note, some statements, disclaimers, and definitions:

1. This is not a substitute for the QuickDraw II introduction in the *Apple IIGS Toolbox Reference*, but rather a supplement.
2. A pixelmap is a series of bytes that hold pixel data whose rectangular shape is defined by a `LocInfo` structure.

This Note describes in great detail the way that QuickDraw II does things with pixelmaps. It begins with a description of the `LocInfo` structure, which is the most important thing to understand in terms of QuickDraw II pixelmap management. Once this is understood, this Note covers how it applies to using functions such as `PPToPort`, `PaintPixels`, and `CopyPixels`. And once this is understood, it then describes how `LocInfo` structures are used to control drawing into a `grafPort`. (`PPToPort` is used in this Note. `PaintPixels` and `CopyPixels` are very close in function to `PaintPixels`. The information and theory in this Note also apply to these calls.)

Understanding the material in this Note should help you better understand the entire toolbox. It is surprising how much can be accomplished with the toolbox without completely understanding these concepts; it is also surprising how much easier programming with the toolbox gets when these concepts are fully understood.

Note: Structures are written with C syntax in this Note. In addition, this Note uses the screen address 0xE12000L. The possibility of shadowing being active and the screen address being 0x12000L is ignored.

The Beginning

One must begin with the `LocInfo` structure, which is as follows:

```
struct LocInfo {
       Word          portSCB;              /* SCB in low byte */
       Pointer       ptrToPixImage;        /* ImageRef */
       Word          width;                /* Width */
       Rect          boundsRect;           /* BoundsRect */
};
```

For this Note, one can change this structure a little bit by calling the `width` element `rowBytes`. This convention is good because `rowBytes` is more descriptive than `width` (it indicates that one is measuring the width in bytes) and it allows one to use the word "width" elsewhere in this Note without confusion. So for the purposes of the Note, the new LocInfo structure definition is as follows:

```
struct LocInfo {
       Word          portSCB;              /* SCB in low byte */
       Pointer       ptrToPixImage;        /* ImageRef */
       Word          rowBytes;             /* Width in bytes*/
       Rect          boundsRect;           /* BoundsRect */
};
```

The `ptrToPixImage` field is a pointer to some block of bytes in memory. (This block of bytes is referred to as the `pixImage` from here on.) A `pixImage` doesn't have any inherent shape. QuickDraw II deals with it as a rectangle, and the `LocInfo` record defines the rectangularity of it.

When saving a 32,000 byte screen image, one doesn't save the number of bytes of which each row consists. One assumes that each row is 160 bytes by convention, and this is a safe assumption, since the IIGS video hardware expects 160 bytes. But the point is that in the 32,000 bytes of screen data, there is no indicator as to the specific size of a row. One must just know that it is 160 bytes per row. This size is fine for screen shots, but it is not fine when different pixelmaps can be different widths. If they can be different widths, then one also needs some information as to what those widths are, hence the `portSCB`, `rowBytes`, and `boundsRect` fields in a `LocInfo` structure.

The `boundsRect` and `portSCB` fields tell the shape of the pixelmap in pixels, the `boundsRect` tells how many pixels wide and tall the pixelmap is, and the `portSCB` tells how big those pixels are (320-mode pixels are four bits wide and 640-mode pixels are two bits wide). One would think that this would be enough information to determine the size of the `pixImage`, but it isn't. The `rowBytes` can be larger than the `boundsRect/portSCB` would indicate (see Figure 1). This situation is legal; it means that some bytes are being wasted, but it is legal.

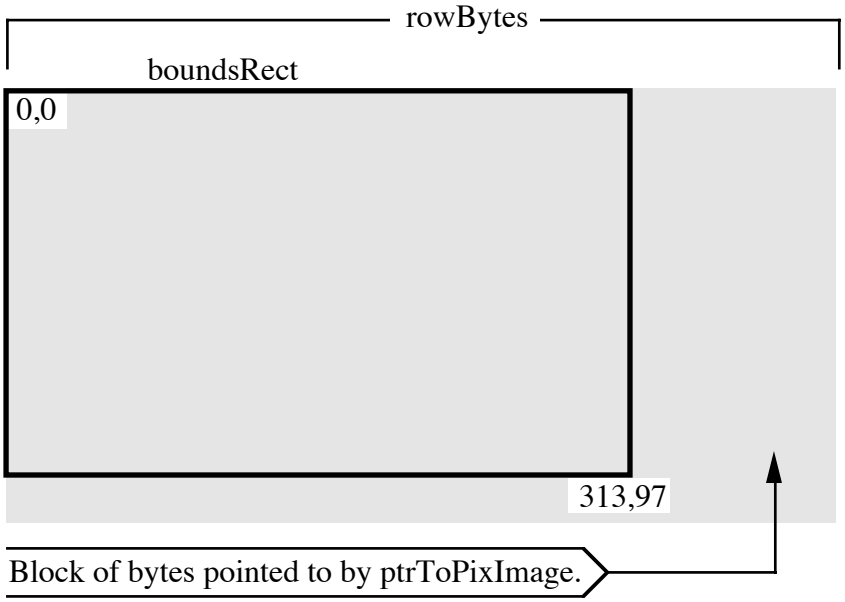

Figure 1–Sample LocInfo Structure

One simply has to know the size of the `pixImage`, since it cannot be determined by the `LocInfo` information. If the `pixImage` is the screen, then it is 32,000 bytes. If it is a fixed or locked handle, then one can do a `FindHandle` on the pointer followed by a `GetHandleSize` on the found handle.

Figure 1 represents a sample `LocInfo` structure. The `portSCB` (although not pictured) is also relevant, as it determines the size of the pixels. If the pixelmap is a 320-mode pixelmap, one could change it to a 640-mode pixelmap by changing the `portSCB` to 640 mode and doubling the width of the `boundsRect`. In doing this conversion, note that `rowBytes` is not affected and that the `pixImage` does not change size.

In the example illustrated in Figure 1, the `pixImage` is bigger than the `boundsRect`, but again, this is okay. However, this is not the case for the screen, where the `rowBytes` is 160 and the height of the `boundsRect` is 200 (the size of the screen is exactly equal to 160 * 200 = 32,000).

There are some rules to determining the `rowBytes` value. First, `rowBytes` must not be too small. This is obvious. Second, `rowBytes` must be evenly divisible by eight. This is not at all obvious, but it is very important. QuickDraw II makes some assumptions for speed, and one of them is that `rowBytes` is a multiple of eight.

So much for describing the `LocInfo` structure. Now for how to use it via `PPToPort`.

`PPToPort` accepts (among other things) a pointer to a source `LocInfo` record and a pointer to a source rectangle. `PPToPort` does not use the source rectangle directly; it first intersects it with the `boundsRect` in the `LocInfo` record, and it uses this intersection rectangle instead. This intersection rectangle guarantees that the area involved is completely enclosed by the

`boundsRect` (and therefore within the `pixImage`). If the source rectangle is entirely outside the `boundsRect`, then the intersection of the source rectangle and the `boundsRect` is empty, thus nothing is drawn.

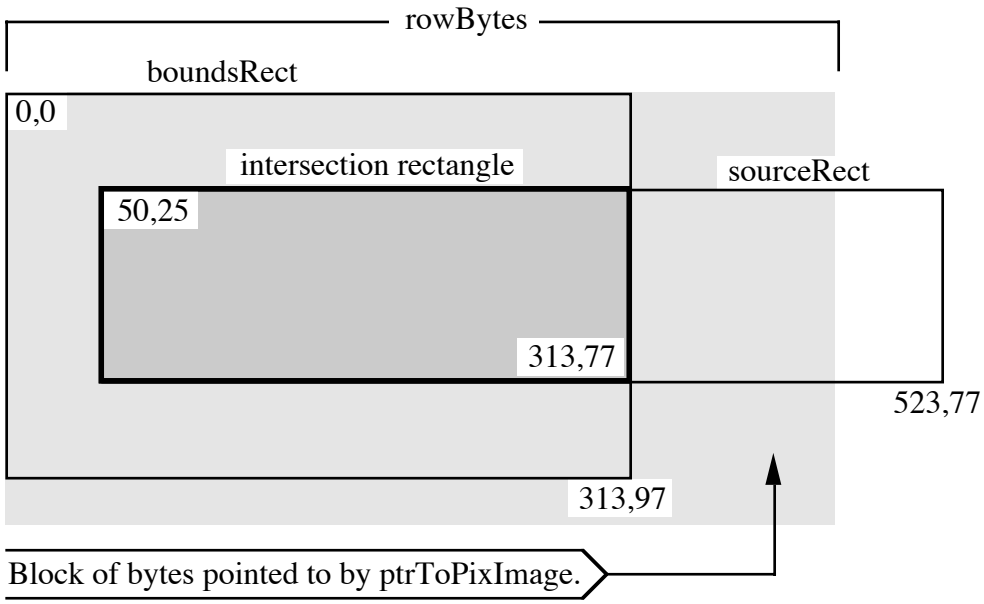

Figure 2–Sample LocInfo Structure With sourceRect

Figure 2 contains a `sourceRect` which is not completely contained by the `boundsRect`; the `sourceRect` is so wide that it even goes beyond the edge of the `pixImage`. If the entire contents of this rectangle were drawn, the result would be quite a mess, since it extends beyond the boundary of the pixelmap. However, `PPToPort` first intersects the `sourceRect` and the `boundsRect`, and then uses the resulting intersection rectangle (illustrated with a thicker border in the figure). `PPToPort` uses only the contents of the intersection rectangle.

Up until now, the boundsRect upper-left corner has always been 0,0. This is an easy way to think of it, but it is not necessary. The important thing to remember about these rectangles is their relation to one another. If one were to offset both the `boundsRect` and `sourceRect` in this example, the values for the corners of the rectangles would change, but the relationship between the two rectangles would stay the same. Figure 3 illustrates the same example if one were to offset both rectangles by -60,-45.

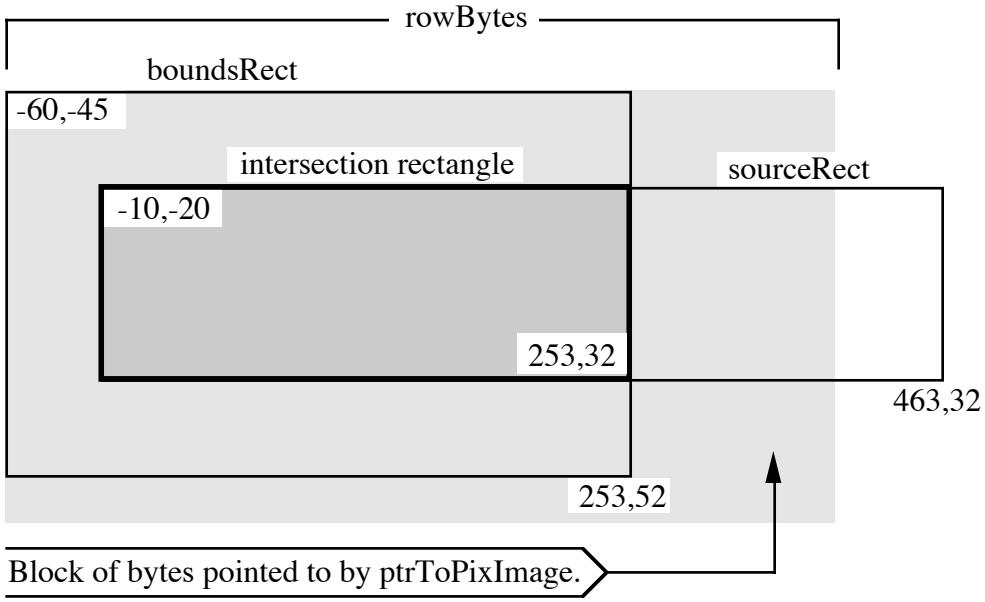

Figure 3–Sample LocInfo Structure Offset by -60,-45

Notice that the same area of the `pixImage` is involved, even though the `boundsRect` and `sourceRect` are offset. When one offsets both the `boundsRect` and `sourceRect` by the same amount, the referenced part of the `pixImage` does not change—this is an important concept.

Time to ask a question that is answered shortly: "Why isn't the upper-left corner of the `boundsRect` always 0,0?" Because the `LocInfo` record isn't always a source `LocInfo` record. It can also be a destination `LocInfo` record, and the most common pixelmap to which a destination `LocInfo` record refers is the screen.

If you had not noticed, the discussion changes gears here—to discuss `LocInfo` records that indicate a destination pixelmap. Basically, everything is the same as has been described with two exceptions. First, destination pixelmaps do not have a `sourceRect`. Instead there is a rectangle that describes some portion of the destination pixelmap, and this rectangle is called the `portRect`. Second, the `LocInfo` record is part of a `grafPort`, and each `grafPort` has a `LocInfo` record as part of the `grafPort` data structure.

It is important to remember that a `LocInfo` record can be used as either a source or destination `LocInfo`. All a `LocInfo` record does is define some bytes in memory as a `pixImage`. Even the screen, which is usually used as a destination pixelmap, can be used as a source pixelmap. There could be situations where one might want to take part of the screen and copy it into some off-screen pixelmap, and in this case, the screen would be a source of pixel data, not a destination.

In the case of the screen pixelmap, there are no wasted bytes in the `pixImage`, as all of the screen bytes are enclosed by the `boundsRect`. The screen width of 160 is evenly divisible by

eight, so there is no slop at the right edge, and there are no extra rows hanging off the bottom of the `boundsRect`.

Figure 4 shows a sample `LocInfo` and `portRect` (every `grafPort` has a `LocInfo` and a `portRect`).

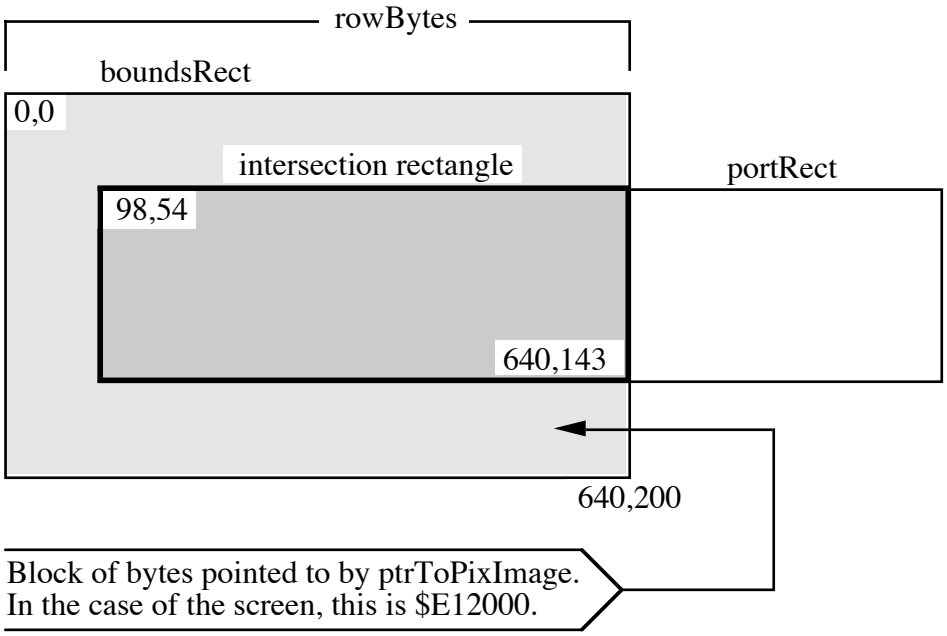

Figure 4–Sample LocInfo and portRect

Following are two important points to remember:

1. Every `grafPort` works in local (not global) coordinates (local coordinates are defined soon).
2. The origin of the `grafPort` is the upper-left corner of the `portRect`. There is no `GetOrigin` call; there is a `SetOrigin` call, but no `GetOrigin`. To get the origin of a `grafPort`, one needs to do a `GetPortRect` call, and then look at the upper-left corner to determine the current origin of the `grafPort`. This is **the** way to get the origin.

In the case of Figure 4, local and global coordinate systems are the same, as is always the case when the `boundsRect` has an upper-left corner of 0,0 (which it seldom does). So, for this exceptional case, one doesn't need a definition of local coordinates. In the global coordinate system, the upper-left corner of the screen is 0,0. In local coordinates, the upper-left corner of the screen is whatever the `boundsRect` says it is. So when the upper-left corner of the `boundsRect` is 0,0, the global and local coordinate systems are the same.

In Figure 4, if one tried to draw something to point 0,0, it would not draw—it would be clipped because it is outside the portRect. So even if one tried to draw there, it would not change point 0,0. If a user moved a mouse to that location and an application performed a `GetMouse` (which

returns the mouse location in the local coordinates of the current `grafPort`), it would return 0,0 as the mouse location.

If one did a `SetOrigin(0,0)`, then the `boundsRect` and `portRect` would be offset by the difference between the old and new origins. Both rectangles would be offset, so the relationship between them would remain the same, as Figure 5 illustrates.

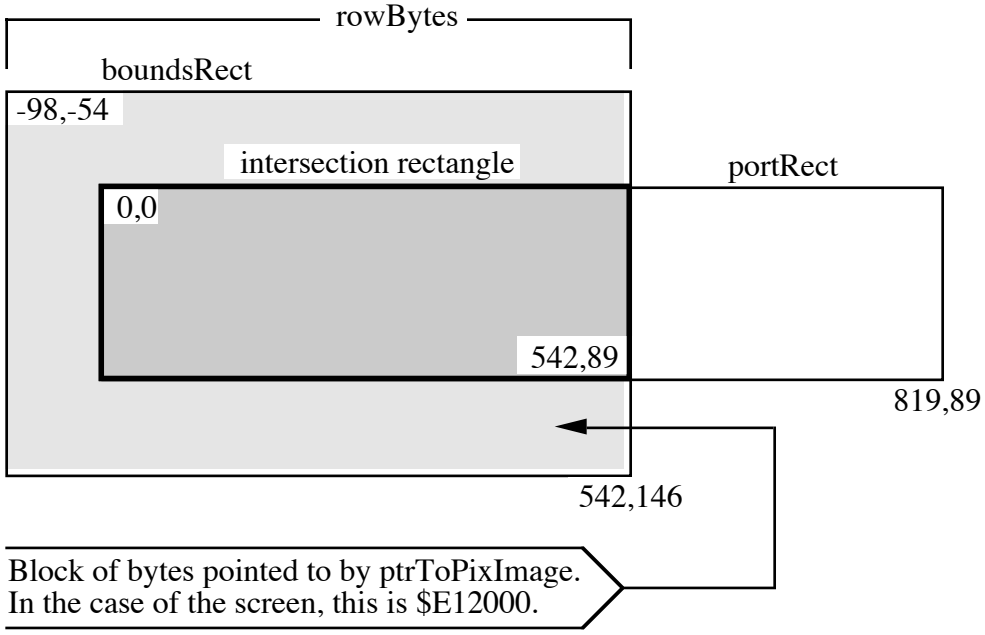

Figure 5–Sample LocInfo and portRect, Both Offset

Now if a user moves a mouse to the upper-left corner of the screen, a call to `GetMouse` returns a value of -98,-54, as expected, and if a user moves the mouse to the upper-left corner of the `portRect`, a call to `GetMouse` returns 0,0, again as expected. This is how origins work and how the conceptual drawing space relates to the `grafPort`. The `boundsRect` of the `grafPort` (in the `LocInfo` record of the `grafPort`) and the `portRect` of the `grafPort` are offset when one calls `SetOrigin`. It is that simple.

Now that it is simple, time to complicate matters with one more player in the QuickDraw II clipping world: the `visRgn`.

The `visRgn` exists for one purpose: to cause more clipping. It never causes anything to be clipped less than the `portRect` does, and in the case of a top window that is completely visible, the `visRgn` and the `portRect` are exactly the same size. Even more than that, the enclosing rectangle for the `visRgn` (every region has an enclosing rectangle) is this case would be exactly the same as that of the `portRect`. This all makes sense when one looks at the purpose of a `visRgn`. Again, the `visRgn` can only cause more clipping. If the entire window is visible, one does not want more clipping, so a `visRgn` the same size as the `portRect` guarantees that it does not clip any more than the `portRect`, as it must clip the same amount.

The `visRgn` is a different size than the `portRect` when the window is not the top window and part of it is overlapped (or if part of the window is off the screen). The part that is overlapped is excluded from the `visRgn`, and this excluded part is clipped to protect the window above from being drawn upon. This is how window clipping works. This is all there is to it.

Figure 6 enhances Figure 5 by adding an overlapping window to demonstrate the `visRgn`.

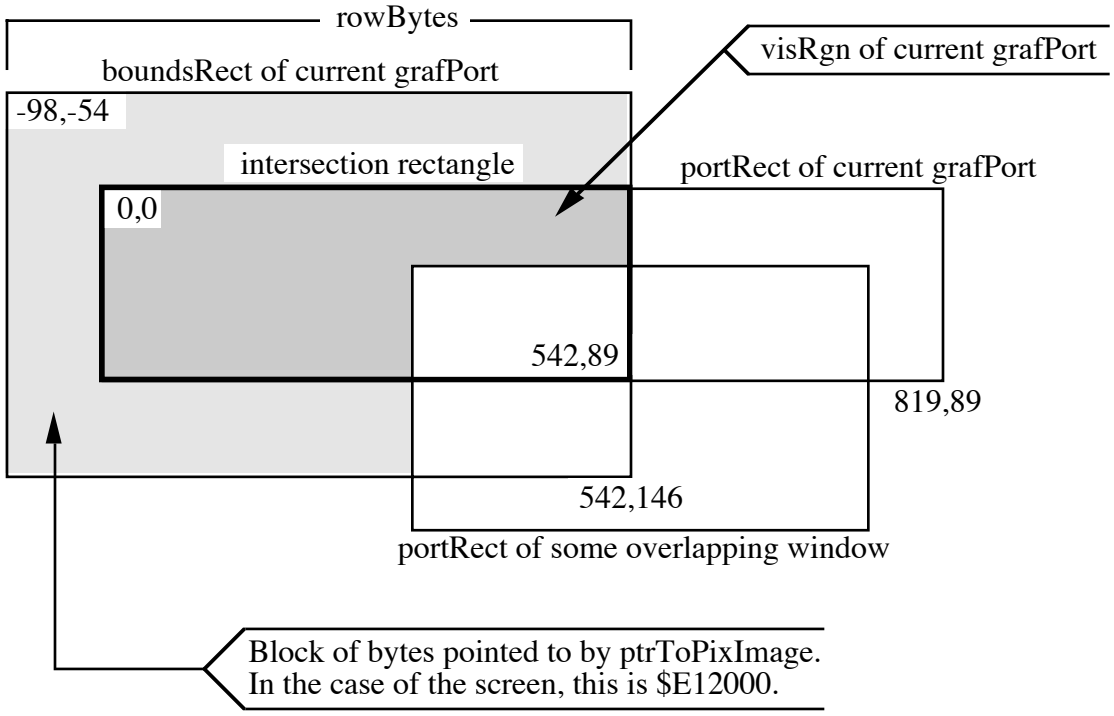

Figure 6–Sample LocInfo and portRect With Overlapping Window

What happens to the `visRgn` during a `SetOrigin`? Remember that the `boundsRect` and `portRect` get offset. The `visRgn` does too. Again, if all of these elements are offset together, then the relationship between them remains the same; they stay the same, relative to one another. (For more information, see Einstein's theory of general relativity.)

The final component for clipping is the `clipRgn`, which is the application's property and, therefore, the application's responsibility. The system sets the `clipRgn` about as big as it can get to start (much bigger than the `portRect`); this is often referred to as arbitrarily large, even though it isn't so arbitrary. The system creates all `grafPort` structures with a large `clipRgn`, and this can be a problem for certain types of QuickDraw II operations. Since the `clipRgn` already reaches to the borders of the conceptual drawing space, it cannot be offset; it is effectively stuck, due to its size. It is a good practice to make the `clipRgn` smaller than the system default.

`SetOrigin` does **not** offset the `clipRgn`. (This is why the size problem with a big `clipRgn` is not so apparent.) The `clipRgn` is the only clipping component that is not offset by

SetOrigin, and one should consider this when using clipRgn for clipping effects, since an application must remember to offset it if it needs to be offset.

Now with all of the fundamentals out of the way, it is time to play some grafPort clipping games. As a refresher, there are four clipping components in a grafPort: the boundsRect, the portRect, the visRgn, and the clipRgn.

If an application creates its own off-screen grafPort structures, then it can do as it wishes with all four clipping components. After all, if it has the responsibility to set them up in the first place, it should have the right to change them. If, however, the Window Manager creates the grafPort structures, then an application should keeps its figurative hands off certain clipping components, namely the boundsRect and the visRgn. The clipRgn, by definition, is the application's to do with as it sees fit, and if careful, an application can also change the portRect. Changing the portRect can be very useful, but one needs to be careful and fully understand all of the ramifications.

So, why would one change the portRect, and how would one do it?

Another figure is in order.

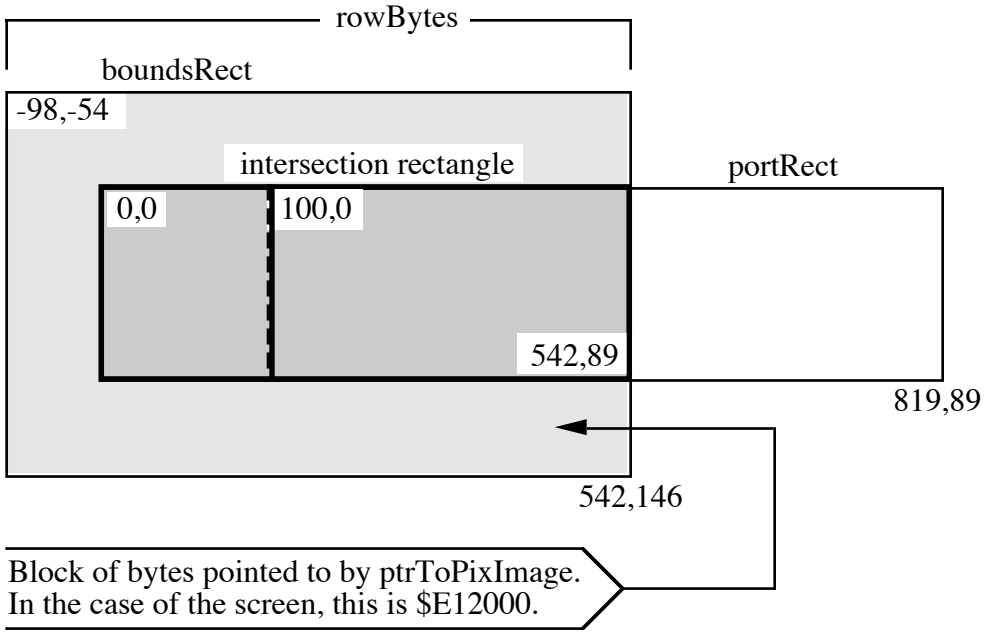

Figure 7–Sample LocInfo and Modified portRect

One can use the GetPortRect call to get the portRect for the current grafPort. One can then modify it, and then use the SetPortRect call to inform the grafPort about the change. Why do this? In Figure 7, the dotted line represents the new left edge of the portRect after the modification (a simple modification of adding 100 to the old value of zero).

Note that changing the `portRect` in this way changes the relationship between the `portRect` and the `boundsRect`. Anything drawn from 0 to 99 (x coordinate) is clipped, since it is outside the new (modified) `portRect`. Before the modification, anything drawn from 0 to 99 would have affected the screen.

This modification may cause the `portRect` to be smaller than the `visRgn`. This is okay, since the `visRgn` can only cause more clipping, not less. So, all of this works just fine. Note that the origin changed when the left edge of the `portRect` changed. The upper-left corner of the `portRect` is always the origin, and an application changed it. The origin changed without a `SetOrigin` call. (Scary, huh?)

One could have done exactly the same thing by making a `clipRgn` to exclude the x coordinates from 0 to 99. However, here is something cool. After the modification, do a `SetOrigin(0,0)`, which sets the upper-left corner of the shrunk `portRect` to 0,0. One cannot accomplish this sort of thing as simply by making a `clipRgn`. One can effectively move where an origin of 0,0 is the screen, and just building a `clipRgn` to exclude some part of the screen does not accomplish this.

Why would one want to change where 0,0 is on the screen? This sort of trick is very useful for adding rulers to a document window, for example. One of the problems with rulers is that they should not scroll with the rest of a document. Unfortunately, `TaskMaster`, if allowed to handle scrolling, doesn't know about a ruler at the top of a window and scrolls it with the rest of the window's content area. By changing the `portRect` so that the ruler is not inside of it, one can keep `TaskMaster` from scrolling it. In a draw procedure, when it is necessary to draw the ruler, grow the `portRect`, set the origin to 0,0, and then draw the ruler. Once it is drawn, set the `portRect` back to the smaller size to protect the ruler again.

Another reason one might want to do this is if an application uses a split window (where the top of the window may show a different part of the document than the bottom). Changing the `portRect` has the advantage that the upper-left corner of the `portRect` is always the origin, so it makes mapping document coordinates easier.

Another advantage to using the `portRect` in this way is that it keeps the `clipRgn` free for other purposes. Being able to separate types of clipping to either the `portRect` or the `clipRgn` keeps the `clipRgn` from being overused.

As a final note, it should be observed that the only clipping that is done is on a destination pixelmap. There is no clipping on a source pixelmap. There is no need. All the clipping needed is done at the destination end, so it would be wasteful to clip twice.

This finishes the discussion about QuickDraw II and how the `boundsRect`, `portRect`, `visRgn`, and `clipRgn` work together to accomplish clipping. Hopefully this Note answers more questions than it creates.

Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2

- *Relativity the Special and General Theory* (1920)

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#81: Extended Control Ecstasy

Revised by: Dave Lyons July 1991
Written by: C.K. Haun May 1990

This Technical Note discusses special features and concerns that should be considered when using the extended controls introduced in System Software 5.0.

Changes since November 1990: Added information on which fields the Control Manager automatically copies from a custom control's template to its control record. Corrected `NewControl2` parameter order. Added a note about `SetCtlTitle`. Changed some "`pea 0`" instructions into "`pha`" when pushing space for results.

Introduction

The extended controls introduced in System Software 5.0 allow the application programmer a great deal more freedom in designing and controlling applications. The new features enhance the functionality of the controls and `TaskMaster,` but can cause confusion and consternation if you are careless with the new parameter block . This Note also includes a discussion of the multipart nature of many of the extended controls and some pointers for writing extended custom controls.

Counting The Costs

One of the major stumbling blocks seen when programming the new extended controls is bad parameter counts. Extended controls introduce parameter blocks and parameter counts to the Control Manager. You need to fully understand the parameters required and the resulting parameter count for each control you create, or you can experience program problems that may be very confusing and difficult to track down.

Remember also that the Control Manager does **not** understand "skipping parameters." If you are creating an extended radio button and you want key equivalents, but not a color table, you **cannot** ignore the color table parameter field. There is no way to tell the Control Manager to "skip" a field during control creation; make sure you initialize all the fields to values that are meaningful—either a real pointer, handle, resource ID, or zeroes. In this case, if you try to "skip" the color table and do not zero out the color table parameter field, the resulting radio button wears ugly colors you do not expect.

As you can imagine, miscounting parameters in other extended controls can produce confusing results, so the parameter count is the **first** place you should check when you're having difficulties creating extended controls.

For Rez users, the `Types.Rez` file contains extended control templates that automatically generate the correct count value, so programmers creating all their controls with the Resource Compiler should not have these problems.

Silly Little Bits

The other area of the new parameter block model that is giving folks trouble is the `moreflags` field. This field hones the definition given by the `reference` fields and can cause you much grief if misused. Make sure to set the reference bits to the values you require. The bit settings have been standardized across all the extended controls, with `%00` indicating a pointer, `%01` indicating a handle, and `%10` indicating a resource. Remember also to set the bits for **all** the references you have—strings, color tables, or whatever else may be ambiguously referenced in the control.

If you accidentally use the wrong bit pattern, you can experience strange bugs ranging from garbled text to `SysFailMgr` caused by a nonexistent resource being referenced. Again, Rez users can use the equates for all the reference specifiers in the Types.Rez file to avoid confusion and evil bugs.

The Parts are Greater than the Whole

To create some new extended controls, like pop-up menu controls and LineEdit controls, functions of different tool sets were combined. Pop-up menu controls are a combination of the Control Manager and the Menu Manager, LineEdit controls are a blending of the Control Manager and the LineEdit tool set, and other new control types follow the same pattern.

This means that, at times, you have to go further into the documentation to find information. Getting the text out of an LineEdit extended control is a multistep process that is a good example of this type of problem.

```
MyLineEdit dc     i2'8'                   ; parameter count
           dc     i4'1'                   ; id number 1
           dc     i2'10,10,23,90'         ; control rectangle
           dc     i4'editLineControl'     ; process reference
           dc     i2'0'                   ; flags
           dc     i2'fCtlCanBeTarget+fCtlWantEvents+fCtlProcRefNotPtr'
                                          ; moreflags
           dc     i2'0'                   ; refcon
           dc     i2'15'                  ; maximum characters allowed
           dc     i4'0'                   ; no default text
MyLineEditHandle ds 4                     ; handle for the created control

           pha
           pha
           pushlong mywindowgrafport      ; window that control will reside in
           pea    0                       ;verb, single Extended control
           pushlong #MyLineEdit
           _NewControl2
```

```
            pla
            sta    MyLineEditHandle        ; save the control handle
            pla
            sta    MyLineEditHandle+2
```

When you want to get the text back out of that control later, you begin to experience what it means to have a control that is an amalgam of various tools. You would start by using the control handle returned by `NewControl2`:

```
Scratch    equ    $0                    ; some scratch space
Scratch2   equ    $4

           lda    MyLineEditHandle      ; move the ControlHandle to
           sta    Scratch               ; some direct page space
           lda    MyLineEditHandle+2
           sta    Scratch+2
           lda    [Scratch]
           tax
           ldy    #2
             lda     [Scratch],y             ; and dereference it, putting it back in
some dpage
           sta    Scratch+2             ; space to use it.
           stx    Scratch
```

That gives you the pointer to the control record. Stored in the control record is the handle of the LineEdit item that is actually controlling the text processing:

```
             pha                                  ; make space for the text handle to be
returned
           pha
           ldy    #octlData             ; offset to the ctlData section
                                        ; of the ControlRecord
           lda    [Scratch],y           ; where the handle for the actual LineEdit
           tax                          ; item was stored
           iny
           iny
           lda    [Scratch],y
           pha
           phx
           _LEGetTextHand               ; ask for the handle for the text
           pla                          ; in this LineEdit control
             sta     Scratch2               ; and now you have the handle to the text
you want.
           pla
           sta    Scratch2+2
```

The main point is that when you are using extended controls, you often cannot use the Control Manager to do everything that needs to be done. You also need to understand and use the supplementary or "hidden" tool sets.

Here's another example, using a pop-up menu extended control, and in this case we define a font pop-up that contains all the font names currently available.

```
MyPopUpControl dc i2'9'                 ; parameter count of 9
           dc    i4'1'                  ; control ID of 1
            dc    i2'2,2,0,0'             ; Position, upper left corner of the window,
let
                                        ; Control Manager calculate full size
           dc    i4'popUpControl'       ; def proc for PopUp
           dc    i2'0'                  ; flags
           dc    i2'fCtlWantEvents+fCtlProcRefNotPtr'
                                        ; more flags
           dc    i4'0'                  ; ref con
           dc    i2'0'                  ; title width, will be calculated
           dc    i4'mymenu'             ; pointer to actual menu structure
```

```
            dc    i2'500'                ; initial value, item number of item
                                         ; to be displayed in popup at creation
```

```
mymenu     dc    i2'0'                    ; version number, should be 0
           dc    i2'200'                  ; menu ID number
           dc    i2'0'                    ; menu flags
           dc    i4'mymenutitle'          ; pointer to menu title
           dc    i4'mymenuitem1'          ; first menu item
           dc    i4'mymenuitem2'          ; second menu item
           dc    i4'0'                    ; null terminator, end of menu
mymenutitle str 'Font'

mymenuitem1 dc i2'0'                     ; version number
           dc i2'500'                    ; item number
           dc i2'0'                      ; no hot keys
           dc i2'0'                      ; not checked
           dc i2'0'                      ; item flags, no special drawing
           dc i4'mymenuitem1title'
mymenuitem1title str 'Plain'

mymenuitem2 dc i2'0'                     ; version number
           dc i2'501'                    ; item number
           dc i2'0'                      ; no hot keys
           dc i2'0'                      ; not checked
           dc i2'1'                      ; item flags, bold face this one
           dc i4'mymenuitem2title'
mymenuitem2title str 'Bold'
```

Now create this control:

```
           pha
           pha
           pushlong mywindow             ; target window grafptr
           pea    0                      ; verb, single control pointer
           pushlong #MyPopUpMenu
           _NewControl2
           pulllong mypopuphandle        ; save the handle
```

This pop-up menu control created is **not** associated with the menu bar across the top of the desktop. You can consider each of your pop-up menu controls as separate menu bars, so if you want to perform Menu Manager calls on a pop-up menu control, you need to set the menu to point at your pop-up menu control. In this example, to add all the fonts available to the pop-up menu you would:

```
           pha
           pha                           ; space to hold current bar
           _GetMenuBar                   ; get the handle to the current menu bar
           pushlong mypopuphandle
           _SetMenuBar
           pea    200                    ; id number of this menu
           pea    502                    ; first font family ID number to use
           pea    0                      ; fontspecbits
           _FixFontMenu
           pea    0
           pea    0
           pea    200
           _CalcMenuSize                 ; re-size the popup menu
            _SetMenuBar                     ; restore the previous menu as the current
menu
```

Controls That Are Not Controls

The new picture extended control is not a "full-fledged" control; it has been provided to simplify your programming tasks. The picture control does **not** support normal mouse hit testing and highlighting. Think of it as a built-in extension to your content drawing routine, and not as a control. It is provided to allow you to refresh your whole window with a single `DrawControls` call, instead of drawing the controls and then drawing pictures. The icon button extended control has been provided as the graphic full-function control. If you need or want a fully functional control that uses a picture, you should consider writing your own custom control procedure.

Custom Extended Controls

Custom controls can also benefit from all the advantages of extended controls. You can create a custom control that uses a template, can be a resource, has a definition procedure that is a resource, and responds to all the new control calls. If you write an extended custom control or upgrade a previously-written custom control, there are new messages and changes to existing messages of which you need to be aware. These changes are documented in volume 3 of the *Apple IIGS Toolbox Reference*.

The Control Manager copies the following fields from the control template to the control record before it sends your control the `init` message: `ctlOwner`, `ctlID`, `ctlRect`, `ctlFlag`, `ctlHilite`, `ctlMoreFlags`, `ctlVersion`, `ctlRefCon`, and `ctlProc`. The `ctlNext` field is owned by the Control Manager. If any additional fields need to be set up based on the control template (such as `ctlValue`, `ctlData`, `ctlColor`, and any custom fields), your `init` routine needs to take care of it.

Putting your custom control definition procedure in a resource can significantly enhance the functionality of the custom control. You may find it easier to add to all of your programs and you do not have to manage the code space required. If you do write a custom control definition procedure and want to store it as a resource, here are some hints for success.

First, the code you store in your resource fork must be fully compiled and linked code. The code resource converter uses the System Loader to load the code, so the code must be executable code, not object code.

Second, set the `convert` and `locked` bits of the resource attributes for your code resource. The `convert` bit must be set to tell the Resource Manager to call the code resource converter when it loads this resource. The resource type for control definition procedures is `rCtlDefProc`, $800C.

By setting `locked` but **not** `fixed`, memory fragmentation is reduced (because of how the code resource converter and Memory Manager work). Setting the `locked` attribute is also recommended for compatibility with future system software.

Third, keep in mind that this definition procedure may be purged and reloaded whenever the Memory Manager needs the space. This means that you cannot store any information in your

definition procedure if you want to keep track of it between calls to the definition procedure. If you do, and your definition procedure gets purged and reloaded, you lose that data.

If you need data space for your custom control, use the control record as your stash. You can easily either use the fields already provided in the control record, or you can expand the control record to as much space as you need (within sensible limits) and store your data there.

Warning: Control definition procedures are initially loaded with purge level zero. When they are released, they are given purge level three. If they are then reloaded, the Resource Manager does not change the purge level back to zero—your definition procedure may then be purged (even while executing) unless its handle is locked. The solution is to lock your definition procedure handle within the procedure:

```
myPosition    pea    0               ; space for result
             pea    0
             pushLong #myPosition
             _FindHandle
             _HLock
```

and unlock your handle with `HUnlock` on exit. This keeps your procedure safe, while not creating "code islands," which clog up memory.

Changing a Control's Title

If you call `SetCtlTitle` to give a control a new title, everything is great if the new title is referenced the same way as the current title (by pointer, by handle, or by resource ID). If the new title is referenced differently, you must first call `SetCtlMoreFlags` on your control so that the `SetCtlTitle` value can be interpreted correctly.

Conclusion

The extended controls provided in System Software 5.0 and later are a great leap forward for programmers. They relieve the application of much of the tedious detail code that relates to housekeeping, not the guts of application programming. Used in combination with the enhanced `TaskMaster`, you can have an application's visual interface up and running a lot faster, leaving you more time to work on the heart of your application.

Further Reference

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

Apple IIGS
#82: Controlling the Control Manager

Revised by: Matt Deatherage November 1990
Written by: Dave Lyons May 1990

This Technical Note describes an anomaly in the `NewControl2` call in System Software 5.0.2 and provides a solution.

Changes since May 1990: Noted that System Software 5.0.3 fixes this anomaly.

This Note formerly advised of a problem with `NewControl2`—the current port was not set before adding the controls, which gave unpredictable results. System Software 5.0.3 and later fix this problem.

Apple IIGS
#83: Resource Manager Stuff

Revised by: Matt "Even less of a middle name" Deatherage May 1992
Written by: Dave Lyons May 1990

This Technical Note answers your miscellaneous Resource Manager questions.

Changes since December 1991: Added several notes pertaining to System Software 6.0 and a note about making Resource Manager calls from a resource converter. Added new discussion about how "changed" is really a resource attribute.

UniqueResourceID

In System Software 5.0.4 and earlier, calling `UniqueResourceID` with an `IDRange` value of $FFFF does not work reliably. It sometimes returns a system-range ID ($07FFxxxx) if there are already system-range resources of the specified type present in the current search path.

If you are using a development utility that generates resource IDs using `UniqueResourceID`, check the results to make sure no system-range resource IDs are being used by accident. This problem is fixed in System Software 6.0.

What SetCurResourceFile Does

`SetCurResourceFile` is documented in Chapter 45 of the *Apple IIGS Toolbox Reference*, Volume 3 (see especially "Resource File Search Sequence" near the beginning of the chapter).

This explanation might make you think `SetCurResourceFile` rearranges the search path, but it does not; instead, it just makes searches start at a different place in the path. `SetCurResourceFile` is useful for controlling what resource files are searched, not for changing the search order.

How the Toolbox Uses Resources as Templates

The toolbox uses several types of resources as templates for creating other objects. Examples include `rControlList`, `rControlTemplate`, and `rWindParam1`. The toolbox automatically releases these resources from memory as soon as it is through with them, so there is no need to create your template resources with special purge levels in an effort to free more memory. It is not a problem.

Using Resources From Window Update Routines

In System Software 6.0 and earlier there is no special code to set the current resource application when the system calls an application window update routine (See Apple IIGS Technical Note #71 for notes on NDAs and the current resource application).

To avoid a situation where a window update routine cannot get needed resources, obey the following rules:

1. Application window update routines must **either** (a) assume that the resource application has the same value it had when the window was created, or (b) save, set, and restore the current resource application, using `GetCurResourceApp` and `SetCurResourceApp`.

2. NDAs that start the Resource Manager must not call application window update routines, and they must not cause application window update routines to be called (for example, if an NDA calls `TaskMaster` to handle a modal dialog or movable modal dialog, the `tmUpdate` bit in `wmTaskMask` must be off).

CurResourceApp in InfoDefProcs and Custom Windows

The current resource application has no guaranteed value when an information bar definition procedure or custom window definition procedure gets control. These must always save, set, and restore the current resource application using `GetCurResourceApp` and `SetCurResourceApp`.

StartUpTools Opens Resource Forks Read-Only

When `StartUpTools` opens your application's resource fork, by default it opens it with read-only access. If your application needs to make changes to the resources on disk in System Software 5.0.4 and earlier, you need to close the fork and reopen it with read and write access. To close it, use `GetCurResourceFile` and `CloseResourceFile`; to reopen it, use `LGetPathname2` and `OpenResourceFile`.

Note: You must update the `resFileID` field in the `StartStop` record if you close and reopen your resource fork. `CloseResourceFile` disposes the handles of any resources in memory from the file you're closing, so you must call `DetachResource` on any resources you need to keep. (If you pass an `rToolSTartup` resource to `StartUpTools`, the system detaches it for you automatically.)

In System Software 6.0 and later, setting bit 3 ($0008) of the `startStopRefDesc` tells the Tool Locator to open your resource fork with all allowed permissions instead of with just read permission.

Calling StartUpTools From a Shell Application (File Type $B5, EXE)

In System Software 5.0.4 and earlier, `StartUpTools` tries to open the current application's resource fork. It determines the pathname of the "current application" by examining prefix `9:` and making a `GET_NAME` GS/OS call, but do not assume it will always construct the pathname this way. If you call `StartUpTools` from a shell application and expect it to open your EXE file's resource fork, you will be disappointed.

If GS/OS has launched your application, life is good—usually, though, a shell has loaded your shell application directly, so `GET_NAME` returns the name of the shell instead of the name of your application file.

To open your shell file's resource fork, call `ResourceStartUp`, get the pathname by calling `LGetPathname2` on your user ID, and pass the pathname to `OpenResourceFile`. `StartUpTools` uses this strategy all the time in System Software 6.0 and later, meaning you don't have to.

What's NIL in a Resource Map?

The resource maps for open resource files are kept in memory, and the structure is defined in chapter 45 of *Apple IIGS Toolbox Reference*, Volume 3.

The `resHandle` field of a resource reference record (`ResRefRec`) is defined as "Handle of resource in memory. A `NIL` value indicates that the resource has not been loaded into memory." In this case, `NIL` means that the middle two bytes of the four-byte field are zero. In other words, a `NIL` entry in the resource map may have a non-zero value in the low-order byte.

LoadResource and SetResLoad(FALSE)

When you call `LoadResource` on a locked or fixed resource and `SetResLoad` is set to `FALSE`, you may get Memory Manager error $0204 (`lockErr`), because the Resource Manager tries to allocate a locked or fixed zero-length handle, which the Memory Manager does not permit.

Adjusting the Search Depth

If you wish to add some resource files to the beginning of a resource search path and adjust the depth so that the end point of the search is unchanged, it's tempting to use `SetResourceFileDepth(0)` to get the current depth, add one, and set this new depth with `SetResourceFileDepth`.

The problem is that the search depth is often -1 ($FFFF), meaning "search until the end of the chain." If you add your adjustment to -1, you do not usually get the intended effect. A solution is just to check for $FFFF and not adjust the depth in that case.

CurResourceApp after ResourceShutDown

After a `ResourceShutDown` call, the current resource application is always $401E. (The Resource Manager starts itself up at boot time with its own memory ID, $401E. Do not ever call ResourceShutDown while the current resource application is $401E.)

Restoring the CurResourceApp

If you need to start up and shut down the Resource Manager without disturbing the current resource application, call `GetCurResourceApp` **before** `ResourceStartUp`, and call `SetCurResourceApp` to restore the old value **after** `ResourceShutDown`.

It does not help to call `GetCurResourceApp` after `ResourceStartUp`, since the application just started up is always the current resource application.

Shell programs which start the Resource Manager need to call `SetCurResourceApp` after regaining control from a subprogram (for example, an EXE file) which may have started and shut down the Resource Manager, leaving the current resource application set to $401E instead of the shell's ID.

Shell programs that do not start the Resource Manager have nothing to worry about. In this case the current resource application is normally $401E, so when a subprogram calls `ResourceShutDown` life is still wonderful.

What Information is Kept For Each Resource Application?

When you switch resource applications with `SetCurResourceApp`, that takes care of all the application-specific information the Resource Manager has.

There is no need to separately preserve the current resource file, the search depth, the `SetResourceLoad` setting, or any application resource converters that are logged in. All of this information is already recorded separately for each resource application.

"Changed" is a Resource Attribute

This seems obvious when first reading the documentation, but it has a consequence that isn't so obvious.

If you mark a resource as changed with `MarkResourceChanged` and later use `SetResourceAttr` to change that resource's attributes, you must include `resChanged` in the attributes you specify or the Resource Manager does **not** still know the resource has changed.

This means you can undo a `MarkResourceChanged` call, but it also means you need to preserve the `resChanged` bit across `SetResourceAttr` calls if you don't want to accidentally achieve the same effect.

The Resource Manager clears the `resChanged` attribute when a resource is written to disk; the attribute indicates the data in memory is more recent than what's on disk. Normally, adding a resource with `AddResource` sets this bit because the resource isn't actually written to disk until the resource file is updated.

However, if `AddResource` has to make the file longer (by extending the EOF), it writes the resource to disk immediately. This means that in some cases, a resource added with `AddResource` will be properly added but the `resChanged` attribute will **not** be set. Don't be confused if this happens to you.

Making Resource Manager Calls From Resource Converters

Don't. This would be a first-class example of reentrancy, and the Resource Manager is not reentrant in any class.

Who Owns Handles Passed to AddResource?

When you pass a handle to `AddResource`, the Resource Manager is responsible for the handle unless `AddResource` returns an error. Once you call `AddResource`, the handle belongs to the Resource Manager and you must treat it like you would the handle to any other resource.

Named Resource Bugs in System Software 6.0

The new-for-6.0 Resource Manager function `RMFindNamedResource` compares the resource name you requested to named resources incorrectly. The comparison algorithm doesn't compare the lengths of the strings before starting to compare the characters. This means, for example, that if you request a resource named "Raymond" and the Resource Manager encounters a named resource named "Raymond" first, it will return the resource named "Raymond" instead. This anomaly also affects the HyperCard IIGS named-resource XCMD callback functions, even though they don't use the Resource Manager's named-resource calls.

This anomaly also affects `RMLoadNamedResource`, which calls `RMFindNamedResource`.

Debugging Information

The following information is provided for your convenience during program development. It allows you to check exactly what user IDs are using the Resource Manager, what files are in their search paths, and what resource converters are logged in.

Do not depend on this information in your program; it is subject to change in future versions of the Resource Manager.

All the Resource Manager's data structures are rooted in the Resource Manager tool set's Work Area Pointer (WAP). To get the Resource Manager's WAP, call `GetWAP` (in the Tool Locator) with `userOrSystem` = $0000 and `tsNum` = $001E.

The WAP value is a handle to the Resource Manager's block of global data. Several interesting areas in this block are listed below.

+$0A2	curApp	**Word**	Offset into the globals block of the current resource application's Application Record.
+$2B0	sysFile	**Long**	Handle of system file map, or NIL if none.
+$2B4	sysConvertList	**Long**	Handle of system converter list, or NIL if none.
+$2B8	appList	**20*n bytes**	List of Application Records (20 bytes each).

Each Application Record has this format:

+000	appFlag	**Word**	0=entry available, 1=entry used, $FFFF = end of array.
+002	appID	**Word**	User ID of application.
+004	appFiles	**Long**	Handle of application's first resource map, NIL=none.
+008	appCur	**Long**	Handle of application's current resource map, NIL=none.
+012	appConverters	**Long**	Handle of application's converter list, NIL=none.
+016	appReadFlag	**Word**	1=read resources, 0=don't read (`SetResourceLoad`).

+018	appFileDepth	**Word**	Number of files to search in this path.

Converter lists have this format:

+000	n	**Word**	Number of entries in the table (entries can be unused).
+002	theConverters	**6*n bytes**	List of converter entries (6 bytes each).

Each Converter entry has this format:

+000	resType	**Word**	Resource type for this converter ($0000 for unused entry).
+002	convAddress	**Long**	Address of resource converter.

The format for a resource map is described starting on page 45-17 of *Apple IIGS Toolbox Reference*, Volume 3.

Remember, don't depend on this information in your application; use it during debugging, and use it to write debugging utilities.

Further Reference

- *Apple IIGS Toolbox Reference,* Volume 3
- Apple IIGS Technical Note #71, DA Tips and Techniques

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#84: TaskMaster Madness

Written by: C.K. Haun <TR> July 1990

This Technical Note discusses the enhancements made to `TaskMaster` in System Software 5.0.

`TaskMaster` has been expanded to handle extended control actions and give you more information about events in System Software 5.0. This Note discusses some features of the expanded `TaskMaster` and `TaskMasterDA`, and how you can best exploit the new features in your applications.

Stop Making It So Difficult

Developers just want to work too hard. You get a neat new thing like the expanded `TaskMaster`, and you still want to do all the work yourself. The new `TaskMaster` does nearly **everything** for you, as long as you treat it correctly.

What this means is you do **not** have to call `FindControl`, `TrackControl`, `TEIdle`, `LEKey`, handle keystrokes for controls, keep track of click counts, or any of the other mundane event management tasks unless you specifically want to perform actions that `TaskMaster` does not perform. For the standard controls and situations this means that you do not have to do **anything**.

The magic keys to this life of freedom and ease are the five newly defined `taskMask` flag bits, labeled in the interfaces as `tmContentControls`, `tmControlKey`, `tmControlMenu`, `tmMultiClick` and `tmIdleEvents.` This Note looks at what the new bits do for you, but first a word of warning.

Warning: If you set **any** of these new bits, `TaskMaster` assumes you are using the new extended task record. This means that you cannot just go into an older program and set these bits and expect your program to work successfully. You also **must** allocate the additional space for the extended portion of the task record. If you do not, `TaskMaster` puts task data in areas that you do not expect, and Bad Things happen.

Bits 'o This, Bits 'o That

Click Bits

`tmMultiClick` tells `TaskMaster` to keep the new "click information" fields in the extended task record updated. This allows you to have `TaskMaster` keep track of multiclick events; the `wmClickCount` field is one, two or three depending on whether the last action was a single, double, or triple click. In fact, if you can click your mouse button fast enough, you can time quadruple clicks, sextuple clicks, or as high as you want, although anything over triple-clicking is nearly impossible for users to consistently manage. `wmClickCount` just gets incremented by one when the click falls within the double time interval. `wmLastClickTick` is updated with the system tick value at last click. `wmLastClickPt` contains the location of the last mouse click. `TaskMaster` calls `GetDblTime` internally to determine the correct time intervals for these values.

Idle Bits

`tmIdleEvents` tells `TaskMaster` to call the idle routines for controls that need idle events, like TextEdit controls and LineEdit controls. This also means that only the active control is blinking a cursor, since `TaskMaster` is working with the target bits of the extended control records to keep track of which TextEdit or LineEdit control is active and switching the target control in response to mouse clicks and Tab keypresses. This is also the area where you tell `TaskMaster` how to highlight your window controls. Using the Control Manager calls `MakeNextCtlTarget` and `MakeThisCtlTarget` allows you to specify which LineEdit or TextEdit control is active. You can use these calls to highlight input errors the user has made. For example, if someone has entered text in a LineEdit control that requires a number, you can alert the user if he enters non-numeric characters with an `Alert` or `AlertWindow` call. You can then direct the user to the LineEdit control that contains the bad entry by calling `MakeThisCtlTarget` with the handle of that LineEdit control. This deactivates any other target control and moves the insertion point to the LineEdit control that needs the correction.

Contentious Bits

`tmContentControls`, `tmControlMenu` and `tmControlKey` bits are the real workhorses of the expanded `TaskMaster`.

When the `tmContentControls` and `tmControlMenu` bits are set, `TaskMaster` handles the mouse activity side of events—tracking, highlighting or popping-up the selected control. If the control is a radio button, check box, pop-up menu or list control, `TaskMaster` also performs the correct action for the click, either setting the control value, scrolling the list, setting the pop-up menu to the selected item, and so on. `TaskMaster` then returns a `taskCode` of `wInControl` ($21). The control handle is stored in `wmTaskData2`, the part code of the part selected in `wmTaskData3` and the control ID is in `wmTaskData4`. For many of the controls in your windows your application needs to take no further actions, `TaskMaster` has set the control values. When the user closes the window or clicks on a button that causes an action, you can then read the values of all the controls you care about at that point and do what you need to do, instead of keeping track as the user manipulates controls.

The last new bit, `tmControlKey`, works with the `tmControlMenu` bit to handle key events for your extended controls.

When a key event occurs, `TaskMaster` sends the event to the internal routine `TaskMasterKey`. `TaskMasterKey` first looks at the `tmMenuKey` bit (which has been in `TaskMaster` since the Window Manager was implemented). If it is set, then `TaskMaster` tries to handle the event as a menu event, calling `MenuKey` for the current menu bar.

Note: This also means that any key equivalents in your main menu bar (across the top of the desktop) take precedence over key equivalents in your window controls.

If this fails (or that bit is not set) and `tmControlKey` is set, then `TaskMasterKey` polls the controls in the currently open window for any controls that would like this keystroke, either for controls with a `keyEquivalent` field or a pop-up menu control with key equivalents for menu items. If it finds a control that wants the key event, it is handled very much like a mouse event. The action for the control is performed (checking a check box, for example) and the `wmTaskData` fields are filled as they would be for a mouse click, and an event code of `wInControl` ($21) is returned. If a key event did occur, you can differentiate it from a mouse event by looking at the `wmWhat` field of the `taskRecord`. Even though a `wInControl` event code was passed back by `TaskMaster`, the `wmWhat` field is either $0001 or $0003, the former for a mouse down event and the latter if a keystroke stimulated the `wInControl` event.

Even More Bits

All these new features rely very heavily on the changes made to the Control Manager in System Software 5.0. Many of the `TaskMaster` features, keystrokes, target controls, and so on **only** work if you have the `moreFlags` bits set correctly in your control definitions. If you are having difficulty with new `TaskMaster` features, check your control definitions against the information in the Control Manager chapter of Volume 3 of the Apple *IIGS Toolbox Reference* and Apple IIGS Technical Note #81, Extended Control Ecstasy.

Don't Get Goofy

There are some dangers in these new features, of course. By allowing built-in key equivalencies for almost all the controls that can exist in a window, it may be tempting to define key equivalents for everything, and create weird and unusual key combinations for your controls. **Please** remember the *Human Interface Guidelines* (specifically Human Interface Note #8, Keyboard Equivalents) and keep your use of keystroke equivalents to a minimum. Multimodifier keystrokes (Command-Option-Shift, for example) do not enhance the user's experience and can be very confusing.

NDAs Can Have Fun Too

`TaskMasterDA` has also been added to the Window Manager, providing your new desk accessories (NDAs) with the same kind of `TaskMaster` support your applications have. This lets you easily use extended controls inside NDAs, following the same basic rules as in an application. There are only a few things to worry about.

What Does That Stack Picture *Really* Mean?

The input to `TaskMasterDA`, as shown in Volume 3 of the Apple *IIGS Toolbox Reference*, is as follows:

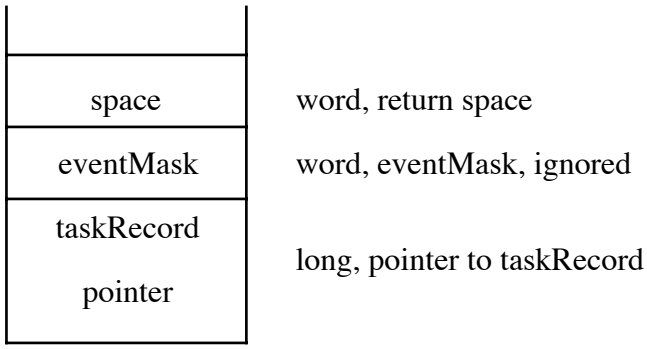

Figure 1–TaskMasterDA Stack Picture

The call returns a word value, the `taskCode`. The space and `eventMask` are self-explanatory. The book tells you that the `eventMask` is ignored, which makes sense since the host application has already gotten the event and you have already specified an `eventMask` in your NDA header, so you can use any value here. The `taskRecPointer` causes the confusion.

You do not pass a blank event record. When your NDA's action routine is called, the `Y` and `X` registers contain a pointer to the current event record with which the NDA is working. `TaskMasterDA` is filling that `taskRecord` with some information, so you want to move it into your NDA's data area so you can work with it later:

```
phy
phx                     ; push the pointer that was passed to us
pushlong  #NDArecord    ; the space in my NDA for the extended event record
pea       0
pea       16            ; only 16 bytes, the original taskRecord size
_BlockMove
```

It is **very** important that you only move 16 bytes. `TaskMasterDA` can act erratically if the extended portion of the event record has been filled with nonsense values. This can happen if your NDA is running in an application that does not use the extended task record and you are copying non-task data into the extended portion of the task record. By the way, as you are debugging your NDA and you run into situations where the `wmTaskData` field values are weird, this is more than likely the problem.

Also remember to make sure the `wmTaskMask` field in your NDA's `TaskRecord` is set and the extended portions of the `TaskRecord` are zeroed out before your NDA starts running; you want to set all these fields in your NDA's `INIT` routine.

Now you can call `TaskMasterDA`:

```
pea       0             ; return space
pea       $FFFF         ; eventMask, ignored
```

```
pushlong  #nDArecord    ; our NDA event record
_TaskMasterDA
pla                     ; event code returned
```

You can then process the event in a convenient way. Remember again that `TaskMaster` has already done the control tracking for the controls in your NDA window. You have the same multiclick information, control handles and IDs.

Conclusion

`TaskMaster` is a wonderful thing that makes any programmer's job easier. So let it work **for** you. Learn the capabilities of the new fields and new controls, and take advantage of them. Let `TaskMaster` take care of the system details, while you concentrate on the features that make your application special.

Further Reference

- *Apple IIGS Toolbox Reference*, Volumes 1 through 3
- Apple IIGS Technical Note #81, Extended Control Ecstasy
- Human Interface Note #8, Keyboard Equivalents

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#85: Moving the Mouse

Written by: Matt Deatherage July 1990

This Technical Note discusses moving the cursor on the screen without touching the mouse.

It is sometimes desirable to programmatically move the QuickDraw II cursor on the screen without requiring the user to touch the mouse. This can be effective, for example, in tutorial software that actually shows mouse actions such as pulling down menus and dragging windows.

There is not an easy or obvious way to do this in the toolbox. This is not a bad thing; it prevents overzealous programmers from zapping the mouse all over the screen for suspicious reasons. You must remember that the mouse belongs to the **user**, not to the application. If the user has put the mouse somewhere, it should **only** be a user's action that causes the cursor to move elsewhere. Most of the time that action is touching the mouse and physically moving it. Do **not** move the mouse except in response to a user-initiated command.

The most obvious way to move the mouse position—calling `PosMouse` with the new mouse position—is not sufficient; `PosMouse` does not update the current mouse position. When the mouse is next moved, a mouse interrupt comes through and the new deltas are added to the old mouse position, resulting in correct `ReadMouse` results **after** the mouse has been physically moved. Also, `PosMouse` does not update the cursor on the screen.

Faking Out the System

When you wish to move the mouse yourself, you are in effect replacing (or adding to) the standard mouse with a small programmatic mouse substitute—your code. This qualifies as a "device" and can be considered an Event Manager "device driver." You can then make the appropriate Event Manager call, `FakeMouse`. When calling `FakeMouse`, you supply all the mouse information yourself, allowing you to move the mouse, simulate button presses, and in general replace the mouse.

Further Reference

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

Apple IIGS
#86: Risking Resourceful Code

Revised by: Matt Deatherage March 1991
Written by: C.K. Haun <TR> September 1990

This Technical Note covers considerations you need to keep in mind when using code resources.
Changes since September 1990: Now lists XCMD and XFCN resources as "Apple's code" and notes that other restrictions apply to them as well.

Code resources are wonderful things that can make your life better than it ever was before. Code resources are necessary when writing CDevs and can be very useful for control definition procedures, code modules for extensible programs like resource editors—in fact, almost anywhere where you use regular compiled and linked code. But to do it right, you need to keep some rules in mind.

Apple's Code, Apple's Rules

The first code resources covered are the ones defined as fully supported by the System Software. These are `rCtlDefProc` ($800C), `rCodeResource` ($8017), `rCDEVCode` ($8018), `rXCMD` ($801E) and `rXFCN` ($801F). Before looking at the specifics, this Note describes in general terms what happens when the Resource Manager loads a code resource.

When you call the Resource Manager with a request for a code resource (or when the system does, as with `rCtlDefProc`s), it loads it like a normal resource. The Resource Manager finds the resource in a resource map in the current search path, allocates a handle for the resource using the attributes in the resource attribute bits, and loads the resource into memory.

Now the Resource Manager examines the `resConverter` bit in the resource header. If this bit is set, indicating that this resource needs to be converted (as it should be for an `rCtlDefProc`), the Resource Manager checks its tables to see if a resource converter has been logged in (with the `ResourceConverter` call). For code resources, the correct converter has been logged in by the manager associated with that resource type. For example, the Control Manager logs in the code resource converter for `rCtlDefProc` resource type.

For code resources, `InitialLoad2` is used to load the OMF from memory. Then the Resource Manager returns a handle containing a pointer to the start of the loaded, relocated code.

Rule 1: Code resources must be smaller than 64K

The code resource converter uses the `InitialLoad2` function of the System Loader to load and convert code resources. That means that code resources are restricted in the same way that loading from memory is. One of these restrictions is that the code must be 64K or less.

Rule 2: Compiled and linked code only

Again, since `InitialLoad2` is used to convert the code resource, the data must be in OMF format since `InitialLoad2` expects to relocate standard load segments. When you prepare your code for inclusion in a code resource, compile and link the code as you normally would for a stand-alone program. Use the file produced by the linker for inclusion in your resource fork. You can use Rez to move the code from a data fork to your application's resource fork with a line in your resource description file similar to the following:

```
read rCodeResource (MyCodeIDNumber,locked,convert)  "MyCompiledAndLinkedCode";
```

Rule 3: One segment please

Multiple segments are theoretically possible with code resources, but you have to manage memory IDs and the memory that the additional code segments use yourself. Since the code resource converter calls `InitialLoad2`, it uses the Memory Manager ID for the current resource application, and you cannot specify a different user ID directly. By changing your current resource application ID (by making an additional call to `ResourceStartUp` with a modified master ID, for example) you could manage multisegment code resources.

Rule 4: No dynamic segments

The dynamic segment mechanism does not work with code resources. Of course, your application can still use dynamic segments, but not code resource dynamic segments.

Rule 5: Set the right attributes

There are two sets of attributes you need to be concerned about for a code resource. The first set includes the standard resource attributes; the second set covers the attributes that the code itself has in the OMF image.

You need **both** sets to get the functionality you want. The resource attributes determine how the Resource Manager handles the resource. The OMF attributes control what `InitialLoad2` does when it converts your code from OMF in a resource handle to relocated executable code.

Remember, you need to set **both** sets of attributes.

The resource attributes you need to set are `locked` and `convert`. The `locked` flag is necessary to prevent the resource from moving while `InitialLoad2` processes it, and the `convert` flag is needed to signal the Resource Manager to call the code resource converter.

You must set the static OMF attribute, the others (like no special memory) you set as appropriate for your code in your application.

Rule 6: Know where to go

The handle you get back from the Resource Manager when you load and convert a code resource points to the beginning of the relocated and ready-to-execute code, **not** to the image of the code that is stored in the resource fork. So you can immediately jump to this code to execute it.

You can override this if you like—clear the `resConverter` bit in the resource attributes. If this bit is zero, the Resource Manager does not call any resource converter (including the code resource converter).

Rule 7: Remember the Write

Keep in mind that any resource that uses a converter uses that converter both for reading and writing the resource. If you write out a code resource, the Resource Manager calls the `Write` routine for the code resource converter, which currently writes without doing any conversion—it does **not** reconvert the code in memory back to OMF format. However, some converters (perhaps one you write) could reconvert the resource before writing it out.

Your Code, Your Rules

If you want to define your own code resource type (with a resource type of less than `$8000` and greater than 0) you may want to follow the same rules as the system code resources use. In fact, you can even use the same code resource converter, by using the `ResourceConverter` call with your resource type, and log the code resource converter as the converter to use with your resource type, like the following:

```
       pha
       pha                     ; return space
       _GetCodeResConverter    ; Misc Tools call to return the loader relocation code
pointer
*                              ; (leave it on the stack for the next call)
       pea $0678               ; resource type you want to convert with this
converter, any
*                              ; Application type you wish
       pea %01                 ; add this converter to the Application converter list,
*                               ; and log this routine in
       _ResourceConverter
```

or you can do whatever you like with the resource, including not having a converter and doing all the relocation and memory management of the code yourself. This can give you the ability to add more functionality than the standard code resources provide—dynamic segmentation is one feature you could implement if you want to handle all the details yourself.

Or, you can manage the code any way you want, but keep the built-in system functions in mind, and use as many of them as you can. Make your life simpler.

One Final Note

If one of your resources is marked `convert` and `preload` the Resource Manager **only** preloads that resource if the converter for that resource is logged in as a converter for that type. If the Resource Manager cannot find the converter, it does not preload the resource.

Further Reference

- *Apple IIGS Toolbox Reference*, Volume 3
- *GS/OS Reference*
- *HyperCard IIGS Script Language Guide*
- HyperCard IIGS Technical Note #1, Corrections to the *Script Language Guide*
- Apple IIGS Sample Code #9, Lister

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#87: Patching the Tool Dispatcher

Written by: Mike Lagae and Dave Lyons September 1990

This Technical Note presents the Apple standard way to patch into the Apple IIGS Tool Dispatcher vectors.

This Note presents MPW IIGS assembly-language code which provides the Apple-standard way for utilities to patch *and unpatch* the Tool Dispatcher vectors. If **all** Tool Dispatcher patches follow this protocol, patches can be installed and removed in any order, without ever accidentally unpatching somebody who patched in after the one getting removed.

Using this protocol, each patch begins with a header in a standard form—a form recognizable by these routines (see `PatchHeader`). This way routines (like `RemoveE10000`) can scan through the list of patches and remove one from the middle.

If your patch is going to stay in the system until shutdown, use this standard patch protocol anyway. This way other utilities can still recognize your patch and scan past it to find the next one. This Note is not just to show you a way to patch the tool dispatcher—it's to show you **the** way. If you patch tool dispatcher vectors in any **other** way, you strip other utilities of their ability to remove their patches.

Of course, patching the Tool Dispatcher vectors slows down all toolbox calls, so you shouldn't patch the tool dispatcher without a pretty good reason. If you need to patch a toolbox function, it is usually better to do it by modifying a tool set's function pointer table instead of patching the dispatcher.

The code in this note is specific to the System tool dispatch vectors ($E10000 and $E10004), but the same technique is recommended for the User tool dispatch vectors—just change $E10000 to $E10008, $E10004 to $E1000C, and `ToolPointerTable` to `UserToolPointerTable`.

What Is This Stuff?

This Note presents the following four routines.

`PatchHeader` is the simplest patch function that obeys the protocol. This is where you put your own patch code.

`InstallE10000` installs a patch into the patch chain. For example:

```
pushlong #PatchHeader
jsl InstallE10000
ply
ply                     ;remove the input parameter
bcc noError             ;error in A
```

`RemoveE10000` removes a patch from the patch chain. For example:

```
pushlong #PatchHeader
jsl RemoveE10000
ply
ply                     ;remove the input parameter
bcc noError             ;error in A
```

`CheckPatch` determines whether the specified address is the starting address of a standard patch. For example:

```
pushlong #PatchHeader
jsl CheckPatch
ply
ply                     ;remove the input parameter
bcc validPatch
```

First, here are some comments and global equates.

```
************************************************************************
*
* Patch.e10000 - Routines to patch into the toolbox dispatch
*                vectors at $e10000 and $e10004.
*
* By Michael Lagae
* Software Quality Assurance
* GS Toolbox Test Team
*
* July 18, 1989
*
* Written for the MPW IIGS Assembler -- Version 1.1b1, 4/9/90
*
* Copyright 1989-1990 by Apple Computer, Inc.
*
************************************************************************

       case yes
       machine M65816
       string asis
       msb on
       print on

       export CheckPatch                ; Check for a valid patch header.
```

```
       export InstallE10000             ; Adds a patch into the toolbox vectors.
       export RemoveE10000              ; Removes a toolbox dispatch vector patch.
       export PatchHeader               ; The simplest toolbox dispatch vector patch.

*************************************************************************
* Equates - Various equates required by these routines.
*
versionNumber         equ $0100          ; The version number of this library.

dispatch1             equ $e10000        ; The first toolbox dispatch vector.
dispatch2             equ $e10004        ; The second toolbox dispatch vector.
ToolPointerTable      equ $e103c0        ; Pointer to the active System TPT.
UserToolPointerTable  equ $e103c8        ; Pointer to the active User TPT.

; Error return values from the routines InstallE10000 and RemoveE10000

noError               equ $0000          ; Value returned if no error occurs.
badHeaderError        equ $8001          ; Patch header wasn't valid.
headerNotFoundError   equ $8002          ; Header to remove wasn't in the linked list.
```

`PatchHeader` is the standard shell for the actual patch code. Your code goes in here, at `NewDispatch2`. When you get control at `NewDispatch2`, the function number is in X and there are two RTL addresses on the stack (pushed after the function's parameters).

Your patch code does not care whether the tool call is being made through the $E10000 or $E10004 vector—in either case you get control with two `RTL` addresses on the stack.

```
*************************************************************************
* PatchHeader - Header required of all routines that will be patched
*               into the toolbox dispatch vectors.
*
* Note:  The code between next1Vector and NewDispatch2 must be included
*        for all calls.  The code below NewDispatch2 only needs to be
*        included for patches that want to post patch the calls.
*
PatchHeader proc
       entry next1Vector,next2Vector
       entry dispatch1Vector,dispatch2Vector
       entry NewDispatch1,NewDispatch2

next1Vector                             ; Where dispatch1 should go when finished.
       jml next1Vector                  ; (Filled in by InstallE10000).
next2Vector                             ; Where dispatch2 should go when finished.
       jml next2Vector                  ; (Filled in by InstallE10000).
dispatch1Vector                         ; Holds the JML instruction from $e10000.
       jml dispatch1Vector              ; (Filled in by InstallE10000).
dispatch2Vector                         ; Holds the JML instruction from $e10004.
       jml dispatch2Vector              ; (Filled in by InstallE10000).

anRtl  rtl                              ; An RTL instruction.  Its address will be
                                        ; pushed on the stack for dispatch1 calls.

NewDispatch1                            ; Entry point for dispatch1 toolbox vector.

       phk                              ; Push program bank.
       pea anRtl-1                      ; Push the address of a RTL.

NewDispatch2                            ; Entry point for dispatch2 toolbox vector.

; The following code should be included in the PatchHeader if the patch wants
; to perform post patching.  This code will determine if the call that was made
; actually exists and if it does, post patching can occur.  If the call doesn't
; exist, any pre-call routines can be executed, but the post patching shouldn't
```

```
; be attempted because the dispatcher will remove the second return address from
; the stack, thus not returning to your post patching routines.
; Stack equates for this routine.

aLong  equ $0001                    ; A temporary long value.
oldDP  equ aLong+4                  ; Where the direct page is saved to.
oldTM  equ oldDP+2                  ; Where the tool call number is saved.

       phx                          ; Save the call that's being made.
       phd                          ; Save the current direct page.
       lda >ToolPointerTable+2      ; Get the TPT to determine the number
       pha                          ; of tool sets loaded.
       lda >ToolPointerTable
       pha
       tsc                          ; Set the direct page to the stack.
       tcd
       txa                          ; See if this tool set exits.
       and #$00ff
       cmp [aLong]                  ; Is it larger than the number of tool sets?
       bcs @noCall                  ; JIF this tool set doesn't exist.
       asl a
       asl a
       tay                          ; Now get the pointer to the FPT.
       lda [aLong],y
       tax
       iny
       iny
       lda [aLong],y
       sta aLong+2
       stx aLong
       lda oldTM                    ; Get the function number.
       and #$ff00
       xba
       cmp [aLong]                  ; Compare it to the number of entries in table.
@noCall
       pla                          ; Remove aLong from the stack.
       pla
       pld                          ; Restore the original direct page.
       plx                          ; Recover the tool number.

; At this point the carry flag is set if the tool call doesn't exist and clear
; if the tool call exits.  No post patching must occur if the carry flag is set.

       jmp next2Vector              ; Go to the original $e10004 jump instruction.

       endp

*******************************************************************************
* CheckPatch - Checks the passed toolbox dispatch vector to see if it
*              points to a valid patch.
*
* Input: Passed via the stack following C conventions.
*     newPatchAddr (long) - Address of the patch routine.
*
* Output:
*     If newPatchAddr is a valid patch -
*          Carry clear
*     If newPatchAddr is not a valid patch -
*          Carry set
*
CheckPatch proc

zprtl         equ $01               ; The address for the rtl on our direct page.
newPatchAddr  equ zprtl+3           ; Address of patch (parameter to this routine).

       tsc                          ; Make the stack the direct page after saving
```

```
        phd                     ; the current direct page.
        tcd
```

```
       lda newPatchAddr+2           ; Simple check to check for a valid pointer.
       and #$ff00
       bne BadPatch                 ; Wasn't zero, can't be a valid pointer.

       lda [newPatchAddr]           ; Check for the first JML instruction.
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$04                     ; Check for the second JML instruction.
       lda [newPatchAddr],y
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$08                     ; Check for the third JML instruction.
       lda [newPatchAddr],y
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$0c                     ; Check for the fourth JML instruction.
       lda [newPatchAddr],y
       and #$00ff
       cmp #$005c
       bne BadPatch

       ldy #$10                     ; Check for the rtl and phk instructions.
       lda [newPatchAddr],y
       cmp #$4b6b
       bne BadPatch

       iny                          ; Check for the phk and pea instructions.
       lda [newPatchAddr],y
       cmp #$f44b
       bne BadPatch

       clc                          ; Calculate the address of the rtl instruction.
       lda newPatchAddr
       adc #$000f
       ldy #$13                     ; Check for address of the rtl instruction.
       cmp [newPatchAddr],y
       bne BadPatch

GoodPatch
       pld                          ; Restore the direct page and report
       clc                          ; that it was a good patch.
       rtl

BadPatch
       pld                          ; Restore the direct page and report
       sec                          ; that something was wrong.
       rtl

       endp

*******************************************************************************
* InstallE10000   - Sets the jump vector at $e10000 and $e10004 to point to
*                 the passed new toolbox dispatch vector patch.  This routine
*                 also updates the linked lists so that more than one routine
*                 can be patched into the dispatch vectors.
*
* Input: Passed via the stack following C conventions.
*     newPatchAddr (long) - Address of the patch routine.
*
```

```
* Output:
*     If an error occurred -
*          Carry set, Accumulator contains one of the following error codes:
*               badHeaderError
*     If no error occurred and patch was installed successfully -
*          Carry clear, Accumulator contains zero.
*
InstallE10000 proc

oldPatchAddr    equ $01                  ; Address of existing patch.
zprtl           equ oldPatchAddr+4       ; The address for the rtl.
zpsize          equ zprtl-oldPatchAddr   ; Size of direct page we'll have on the
stack.
newPatchAddr    equ zprtl+3              ; Address of patch (parameter to this
routine).

       tsc                          ; Move the stack pointer to point beyond
       sec                          ; the direct page variables that we'll
       sbc #zpsize                  ; place on the stack.
       tcs
       phd                          ; Save the direct page register.
       tcd                          ; Set the direct page.
       php                          ; Disable interrupts
       sei

       pei newPatchAddr+2           ; Check if patch header is valid.
       pei newPatchAddr
       jsl CheckPatch
       plx                          ; Remove the parameters from the stack.
       plx
       bcc @1                       ; Report the badHeaderError if detected.
       ldy #badHeaderError
       jmp Exit

@1     lda >dispatch1               ; Set up the next1Vector in the new patch.
       sta [newPatchAddr]           ; The JML instruction and low byte.
       lda >dispatch1+2
       ldy #$02
       sta [newPatchAddr],y         ; The middle and upper bytes.

       lda >dispatch2               ; Set up the next2Vector in the new patch.
       ldy #$04
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       lda >dispatch2+2
       ldy #$06
       sta [newPatchAddr],y         ; The middle and upper bytes.

       lda >dispatch1+3             ; See if there's already a patch in dispatch1.
       and #$00ff
       sta oldPatchAddr+2
       pha                          ; High byte of possible header address.
       lda >dispatch1+1
       sec
       sbc #$0011
       sta oldPatchAddr
       pha                          ; Low byte of possible header address.
       jsl CheckPatch
       plx
       plx
       bcs First                    ; JIF this will be the first patch installed.

       ldy #$08                     ; Set up the dispatch1Vector in the new patch.
       lda [oldPatchAddr],y
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       ldy #$0a
       lda [oldPatchAddr],y
```

```
       sta [newPatchAddr],y         ; The middle and upper bytes.

       ldy #$0c                     ; Set up the dispatch2Vector in the new patch.
       lda [oldPatchAddr],y
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       ldy #$0e
       lda [oldPatchAddr],y
       sta [newPatchAddr],y         ; The middle and upper bytes.

       bra PatchIt                  ; Now patch dispatch1 and dispatch2.

First  ldy #$08                     ; Set up the dispatch1Vector in the new patch.
       lda >dispatch1
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       ldy #$0a
       lda >dispatch1+2
       sta [newPatchAddr],y         ; The middle and upper bytes.

       ldy #$0c                     ; Set up the dispatch2Vector in the new patch.
       lda >dispatch2
       sta [newPatchAddr],y         ; The JML instruction and low byte.
       ldy #$0e
       lda >dispatch2+2
       sta [newPatchAddr],y         ; The middle and upper bytes.

PatchIt
       clc                          ; Calculate the address of the new dispatch2.
       lda newPatchAddr
       adc #$0015
       sta newPatchAddr
       xba
       and #$ff00                   ; Mask in the JML instruction.
       ora #$005c
       sta >dispatch2               ; The JML instruction and low byte.
       lda newPatchAddr+1
       sta >dispatch2+2             ; The middle and upper bytes.

       sec                          ; Calculate the address of the new dispatch1.
       lda newPatchAddr
       sbc #$0004
       sta newPatchAddr
       xba
       and #$ff00                   ; Mask in the JML instruction.
       ora #$005c
       sta >dispatch1               ; The JML instruction and low byte.
       lda newPatchAddr+1
       sta >dispatch1+2             ; The middle and upper bytes.

       ldy #noError                 ; Report that all went well.

Exit   plp                          ; Restore the interrupt state.
       pld                          ; Restore the previous direct page register.
       tsc                          ; Restore the stack pointer.
       clc
       adc #zpsize
       tcs
       tya                          ; Value to return.
       beq @noerr
       sec                          ; Report that there was an error.
       rtl
@noerr clc                          ; Report that there was no error.
       rtl

       endp
```

```
*******************************************************************************
* RemoveE10000 - Removes the specified patch from the dispatch1 and dispatch2
*                vectors and updates the linked lists for the remaining
*                toolbox patches.
*
* Input: Passed via the stack following C conventions.
*     patchToRemove (long) - Address of the patch to remove.
*
* Output:
*     If an error occurred -
*          Carry set, Accumulator contains one of the following error codes:
*               badHeaderError
*               headerNotFoundError
*     If no error occurred and patch was removed successfully -
*          Carry clear, Accumulator contains zero.
*
RemoveE10000 proc

patchDispAddr    equ $01                   ; Address of existing patch (and 1 extra
byte).
prevHeader       equ patchDispAddr+5       ; Used to search through the linked list.
zprtl            equ prevHeader+4          ; The address for the rtl.
zpsize           equ zprtl-patchDispAddr   ; Size of direct page we'll have on the
stack.
patchToRemove    equ zprtl+3               ; Address of patch (parameter to this
routine).

       tsc                         ; Move the stack pointer to point beyond
       sec                         ; the direct page variables that we'll
       sbc #zpsize                 ; place on the stack.
       tcs
       phd                         ; Save the direct page register.
       tcd                         ; Set the direct page.
       php                         ; Disable interrupts
       sei

       pei patchToRemove+2         ; Check if patch header we were asked to
       pei patchToRemove           ; remove is a valid header.
       jsl CheckPatch
       plx                         ; Remove the parameters from the stack.
       plx
       bcc @1                      ; Report the badHeaderError if detected.
       ldy #badHeaderError
       jmp Exit

@1     clc                         ; Create the JML instruction that would exist
       lda patchToRemove           ; if the patchToRemove was installed.
       adc #$0011
       sta patchDispAddr+1
       lda patchToRemove+2
       sta patchDispAddr+3
       lda patchDispAddr           ; Mask in the JML instruction.
       and #$ff00
       ora #$005c
       sta patchDispAddr

       cmp >dispatch1              ; Check if the patch to remove is the first
       bne NotFirstOne             ; patch installed.
       lda >dispatch1+2
       cmp patchDispAddr+2
       bne NotFirstOne

       lda [patchToRemove]         ; Restore the Dispatch1 vector.
       sta >dispatch1
       ldy #$02
       lda [patchToRemove],y
```

```
        sta >dispatch1+2
```

```
        ldy #$04                    ; Restore the Dispatch2 vector.
        lda [patchToRemove],y
        sta >dispatch2
        ldy #$06
        lda [patchToRemove],y
        sta >dispatch2+2

        bra NoErr                   ; Everything went well.

NotFirstOne
        sec                         ; Assume that whatever is in dispatch1 is a
        lda >dispatch1+1            ; patch and get the address of its header.
        sbc #$0011
        sta prevHeader              ; Low and middle bytes.
        lda >dispatch1+3
        and #$00ff
        sta prevHeader+2            ; Upper byte of header address.

@loop   pei prevHeader+2            ; Check if it really is a valid header.
        pei prevHeader
        jsl CheckPatch
        plx                         ; Remove the parameters from the stack.
        plx
        bcc @2                      ; Report that the patch that we asked to
        ldy #headerNotFoundError    ; remove wasn't found.
        bra Exit

@2      lda [prevHeader]            ; See if this patch points to patch we want
        cmp patchDispAddr           ; to remove.
        bne @nope
        ldy #$02
        lda [prevHeader],y
        cmp patchDispAddr+2
        bne @nope

        lda [patchToRemove]         ; Restore the next1Vector.
        sta [prevHeader]
        ldy #$02
        lda [patchToRemove],y
        sta [prevHeader],y

        ldy #$04                    ; Restore the next2Vector.
        lda [patchToRemove],y
        sta [prevHeader],y
        ldy #$06
        lda [patchToRemove],y
        sta [prevHeader],y

        bra NoErr                   ; Everything went well.

@nope   ldy #$02                    ; Get the address of the next patch header.
        lda [prevHeader],y
        tax
        lda [prevHeader]
        sta prevHeader
        stx prevHeader+2

        sec
        lda prevHeader+1
        sbc #$11
        sta prevHeader
        lda prevHeader+3
        and #$00ff
        sta prevHeader+2
```

```
       bra @loop                    ; Now check this header.

NoErr  ldy #noError                 ; Report that all went well.

Exit   plp                          ; Restore the interrupt state.
       pld                          ; Restore the previous direct page register.
       tsc                          ; Restore the stack pointer.
       clc
       adc #zpsize
       tcs
       tya                          ; Value to return.
       beq @noerr
       sec                          ; Report that there was an error.
       rtl
@noerr clc                          ; Report that there was no error.
       rtl

       endp

       end
```

Further Reference

- *Apple IIGS Toolbox Reference*
- Apple IIGS Technical Note #73, Using User Tool Sets

Apple IIGS
#88: The Page One Stack in a 16-Bit World

Written by: Dave Lyons September 1990

This Technical Note clarifies the protocol for moving the stack pointer in and out of page one.

On page 13 of the *Apple IIGS Firmware Reference*, under "Save the value of the native-mode stack pointer," there is a code sample showing how to switch **to** the page-one stack by setting the stack pointer to $01xx, where xx is the contents of `EMULSTACK` at $01/0100.

However, the manual does not warn you about moving the stack pointer from page one to another area. When you do that, you must store the low byte of the stack pointer at `EMULSTACK` before moving the stack pointer out of page one. If you do not save the page-one stack properly, interrupt routines or some toolbox calls may destroy a part of the page one stack that you go back to later, expecting that return addresses are still there.

Note: If the auxiliary-memory stack and zero page are in use, you must use $01/0101 instead of $01/0100. See the *Apple IIe Technical Reference Manual*, pp. 153-154

Further Reference

- *Apple IIGS Firmware Reference*
- *Apple IIe Technical Reference Manual*

Apple II
Technical Notes

Developer Technical Support

Apple IIGS
#89: MessageByName—Catchy Messages

Written by: Dan Strnad & Dave Lyons September 1990

This note clarifies `MessageByName` and provides examples of creating and retrieving a named message.

Did You Say You Want To Get A Message?

All you have to do is ask. *Apple IIGS Toolbox Reference*, Volume 3 already tells you how. Here's what the fine print says: with the `createItFlag` set to `FALSE` and the name of the message you are after in the `nameString`, you call `MessageByName`. What's unclear in the manual is that if the message was found, no error is returned, the `createFlag` is returned as `FALSE`, and `messageID` contains the ID you can pass to `MessageCenter` to retrieve the contents of the message. Here's an example of `MessageByName` in use.

The following code **creates** a named message.

```
CreateNamedMessage
            pha
            pha
            pea 1                                   ;create it
            pushlong #MsgBlock
            _MessageByName                          ;function $1701
            pla
            sta myMsgID                             ;keep the ID if you want
            pla                                     ;check the createFlag if you
want
            ...

MsgBlock    dc.w MsgBlockEnd-MsgBlock
            dc.b 28,'XYZ Software:My Cool Product'  ;Pascal-style string
            ... more data goes here
MsgBlockEnd
```

The following code **retrieves** the message.

```
            pha
            pha
            pea 0                                   ;don't create message
            pushlong #MsgBlock
            _MessageByName                          ;function $1701
            ply                                     ;keep id of existing message
            pla                                     ;createFlag (ignore)
            bcs noMessage                           ;carry set if an error occurred
```

```
            pea 2                                   ;MessageCenter action: GET
             phy                                        ;message ID for MessageCenter,
below
            pha
            pha                                     ;space for NewHandle result
            lda #0                                  ;size of handle (0)
            pha
            pha
            ldx MyID                                ;ID for empty
            phx
            pha                                     ;handle attributes (0)
            pha
            pha                                     ;no special location
            _NewHandle
            lda 3,s
            sta mcHandle+2
            lda 1,s
             sta mcHandle                               ;keep a copy of the handle for
later
              _MessageCenter                                ;takes Action, Msg ID, and
Handle

            lda mcHandle+2
            pha
            lda mcHandle
            pha
            phd
            tsc
            tcd
            ldy #2
            lda [3],y
            tax
            lda [3]
            sta 3
            stx 5

* now read data from the message at [3]
            ldy #$xxxx                              ;index past the name string
            lda [3],y
            ...
            pld
            pla
            pla

            lda mcHandle+2
            pha
            lda mcHandle
            pha
            _DisposeHandle

noMessage   ...

mcHandle    dc.l 0
myMsgID     dc.w 0
```

`MessageByName` is available in Tool Locator versions 3.0 and later (System Software 5.0 and later).

Further Reference

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

Apple IIGS
#90: 65816 Tips and Pitfalls

Revised by: Matt "Matt" Deatherage March 1991
Written by: Dave "Dave" Lyons September 1990

This Technical Note presents short 65816 assembly language examples illustrating pitfalls and clever techniques.

Changes since November 1990: Added more explanations about the JSL table and corrected a comment.

Dispatching Through an Address Table

The 65816 has a `JSR ($aaaa,X)` instruction for calling a selected subroutine from a table of addresses, but it has no `JSL ($aaaa,X)` instruction. If you need to dispatch to one of several routines that are not all in the same bank, you need an approach like the following. The idea is to perform a `JSL` to a routine which does a long jump by pushing a three-byte "`RTL` address" on the stack and then doing an `RTL`.

```
            jsl LngJmp          ;go jump to the routine
            ...

  LngJmp    asl a               ;take routine number in A and
            asl a               ; multiply it by 4
            tax                 ;put table index into X
            lda table+1,x       ;get "middle" word of address
            pha                 ; and push it
            lda table,x         ;get low word and
            dec a               ; decrement it by one
            phb                 ;push a single throw-away byte
            sta 1,s             ;store over low two of the 3 bytes
            rtl                 ;transfer control to the routine
table       dc.l routine1       ;table of 4-byte subroutine addresses
            dc.l routine2
            dc.l routine3
            ...
```

This code is correct because `RTL` pulls three bytes off the stack and increments the two low bytes **without** incrementing the high byte.

Note: This approach to a table-based JSL is more flexible than `JML ($XXXX)` because it does not require any fixed-location storage or bank zero space, other than the stack.

On the other hand, the following code is not correct. The approach here is to make a table of addresses *minus* one.

```
        asl a             W
        asl a             R  ;multiply index by 4
        tax               O  ; and put it in X
        lda table+1,x     N  ;get the "middle" word
        pha               G  ; and push it
        lda table,x       !  ;get the low word
        phb               W  ;push a single throw-away byte
        sta 1,s           R  ;store over low two bytes
        rtl               O  ;transfer control to the routine
table   dc.l routine1-1   N  ;table of 4-byte addresses minus one
        dc.l routine2-1   G
        dc.l routine3-1   !
        ...
```

This second sample code fragment fails if any of the routines in the table comes at the first byte of a bank. For example, if `routine1` is at $060000, the address pushed is $05FFFF, and `RTL` transfers control to $050000, not $060000.

Dereferencing Handles Without Direct Page Space

When your code gets called with the `D` register undefined, you must **not** use direct page addressing without setting `D` to a known good value. Preserving and restoring locations on the caller's direct page is **not** reliable, because `D` could be pointing at bytes below the stack pointer (which can be destroyed by interrupts) or even at the $C0xx soft switches (that would make your direct page accesses accidentally fiddle with hardware).

A common way to get temporary direct page space is to point `D` at part of your stack. This following code dereferences a handle stored in the `A` and `X` registers (if the handle is $E01234 and refers to a block of memory at $056789, then on entry `A`=$00E0 and `X`=$1234, and on exit `A`=$0005 and `X`=$6789).

```
        phd                 ;save caller's direct-page register
        pha                 ;push high word of handle
        phx                 ;push low word of handle
        tsc                 ;get stack pointer in A
        tcd                 ;and put it in D
        lda [1]             ;get low word of master pointer (no ",Y"!)
        tax                 ; and put it in X
        ldy #$0002          ;offset to high word of master pointer
        lda [1],y           ;get high word
        ply                 ;remove low word of handle
        ply                 ; and high word
        pld                 ;restore the caller's direct-page register
```

Direct page addressing isn't the only way to address through pointers. Here's the same routine as before, but using the Data Bank register (B) instead of fiddling with D. (Note that handles do **not** have to be in bank $E0 or $E1, although they usually are.)

```
        phb                 ;save caller's data bank register
        pha                 ;push high word of handle on stack
        plb                 ;sets B to the bank byte of the pointer
        lda |$0002,x        ;load the high word of the master pointer
        pha                 ; and save it on the stack
        lda |$0000,x        ;load the low word of the master pointer
        tax                 ;and return it in X
        pla                 ;restore the high word in A
        plb                 ;pull the handle's high word high byte off the stack
        plb                 ;restore the caller's data bank register
```

Emulation Mode Has 65816 Features

You don't have to switch into Native mode just to do an eight-bit operation with long addressing. Most 65816-specific instructions and addressing modes work in emulation mode in approximately the same way they work in eight-bit native mode. See the "Further Reference" for details.

Further Reference

- *Apple IIGS Hardware Reference*
- *Programming the 65816, Including the 6502, 65C02 and 65802* (Eyes and Lichty, 1986, Brady)

Apple IIGS
#91: The Wonderful World of Universal Access

Revised by: Dave Lyons May 1992
Written by: Don J. Brady, Matt Deatherage, & Ron Lichty September 1990

This Technical Note discusses how your applications can be compatible with Universal Access software.
Changes since July 1991: Added caution against reading the keyboard with interrupts disabled.

What's "Universal Access?"

Universal Access is the name given to software components designed to make Apple computers (in this case, the Apple IIGS) more accessible to people who might have difficulty using them. The Apple IIGS is very dependent on graphic objects, a keyboard and mouse; not all people can use these things very easily.

There are several components to Apple's Universal Access software:

- **CloseView**. CloseView magnifies the Apple IIGS screen so that it's more easily seen by those with visual impairments. The hardware screen contains a magnification from two to twelve times as large as the "real" 32K Super Hi-Res graphics screen.

- **Video Keyboard**. Video Keyboard is a New Desk Accessory that emulates a keyboard. A picture of a keyboard appears on the screen; a mouse-down event in any "key" makes Video Keyboard post a key-down event, so you can use a pointing device as a keyboard. ADB hardware is available to allow people to use head gear or other devices instead of mice; Video Keyboard lets these same devices replace the keyboard as well.

- **Easy Access.** Easy Access comes in two parts: Sticky Keys and Mouse Keys. **Sticky Keys** makes the keyboard easier to use for those who have trouble pressing more than one key at a time; while Sticky Keys is activated, modifier keys may be released and still apply to the next keystroke. **Mouse Keys** allows the numeric keypad to be used as a mouse substitute. Sticky Keys and Mouse Keys are included in all ROM 03 Apple IIGS computers. The software versions allow all Apple IIGS computers to provide these functions, and provide additional icon feedback (in the upper right menu bar) for Sticky Keys.

How It Works (Access Nothing and Checks For Free)

Universal Access generally works by replacing Apple IIGS toolbox functions. For example, CloseView patches QuickDraw so you do not draw to the actual screen, but to another buffer that CloseView can then magnify. Video Keyboard patches the Window Manager so that its keyboard window is always frontmost and fully visible (and accessible). Easy Access uses the ADB tools and the Event Manager to alter the way the hardware responds.

Since Universal Access changes the way the tools behave, your applications do not have to work very hard to be accessible to a broad range of physically challenged people. Just by following the rules, you have an accessible application. There are, however, a few guidelines you should keep in mind when designing your programs to make them as accessible as they can be.

Universal Access Compatibility Guidelines

- Don't disable interrupts and then try to read the keyboard. Easy Access on ROM 1 works at the Apple Desktop Bus level—if ADB interrupts are not being serviced, no keypresses will show up at $C000/$C025. Even Reset will not work, so the user may have to power down to regain control of the machine.

- Try to avoid using modal dialogs. Not only do lots of modal dialogs make for a cumbersome interface for everyone, they are especially annoying to those who have to move the mouse to a lot of OK buttons. More importantly, users cannot open NDAs like Video Keyboard while modal dialogs are frontmost.

  Video Keyboard can also be dragged in **front** of modal dialogs. If you are in the habit of using QuickDraw calls to draw items in Dialog Manager modal dialogs instead of creating custom dialog `userItems`, Video Keyboard users can drag the keyboard window in front of your dialog and erase the items (since the only items redrawn are those redrawn by the Dialog Manager's update routine). You can easily test this in all of your dialogs by obscuring each dialog with the Video Keyboard window a piece at a time, then moving Video Keyboard away, to be sure that all areas are completely redrawn.

  Let's say, for example, that you have a custom text item that changes between invocations of the same modal dialog. You might choose to draw the text yourself with `LETextBox2` after creating the dialog with `GetNewModalDialog` but before letting the Dialog Manager handle events with `ModalDialog`:

```
phx                         ; port: hi word from GetNewModalDialog
pha                         ; port: lo word from GetNewModalDialog
_SetPort

lda OurText+2               ; pointer to text to draw in modal dialog
pha
lda OurText
pha
lda OurTextLength           ; Text length
pha
```

```
        pea OurTextRect>>16         ; Text rectangle
        pea OurTextRect
        pea 0002                    ; Text justification (2 = fill)
        _LETextBox2
```

To be Universal Access–friendly, you would, instead, implement a `userItem` routine like the following:

```
; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
DrawDialogText
;
; DrawDialogText draws text pointed to by OurText into the Dialog.
; This userItem routine is called only by the Dialog Manager,
;   when it's implementing/updating the dialog.
; . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

      lda >OurText+2             ; pointer to text to draw in modal dialog
      pha
      lda >OurText               ; (long addressing: data bank unknown)
      pha
      lda >OurTextLength         ; Text length
      pha
      pea OurTextRect>>16              ; Text rectangle
      pea OurTextRect
      pea 0002                   ; Text justification (2 = fill)
      _LETextBox2

      lda 1,s                    ; get return address
      sta 7,s                    ; move to proper location
      lda 2,s                    ;  above input parameters
      sta 8,s

      pla                        ; move stack pointer up
      pla                        ;  to new return address location
      pla
      rtl
```

It will be called as a result of adding a template item like the following to the dialog template (note use of Item Value for the text length, since template Value fields are not used by userItems):

```
   TextTemplate  dc.w 3                           ; ID
   OurTextRect   dc.w TTop,TLeft,TBottom,TRight
                 dc.w UserItem+ItemDisable        ; Type
                 dc.l DrawDialogText       ; Pointer to our userItem routine
   OurTextLength ds.w 1                           ; Text length (cheap place to put
it)
                 dc.w 0000                 ; Item flag
                 dc.l 00000000                    ; Item color
```

Note that this is a simple example of a custom item routine; if you really had custom text that changed from invocation to invocation, you could use the existing Dialog Manager `ParamText` and `longStatText2` item mechanisms.

- Use the Event Manager routines for event information. Do not access any hardware directly or use the lower-level Miscellaneous Tools routines for user event information—you steal that information from Universal Access. For example, use the Event Manager routine `GetMouse` to find the mouse location. Do not use `ReadMouse` or you steal mouse movement information from Universal Access.

- Call `GetNextEvent` or `TaskMaster` often. Long delays between calls do not let NDAs like Video Keyboard get events. If you cannot make these calls, at least call `SystemTask`.

- Do not assume that the hardware location of the screen is $E12000. Universal Access components that manipulate the entire screen (like CloseView) move the virtual screen so the hardware can be used for the magnified screen image.

  To find the screen location, look at the `ptrToPixImage` field in a `grafPort` after calling `OpenPort` (or in your window's window record after `NewWindow`). The image pointer gives the correct location of the screen.

  Assuming the current port is on screen, the following code finds the `ptrToPixImage` value:

```
pha
pha             ;made space for port pointer
_GetPort
phd             ;save direct page location
tsc
tcd             ;port pointer is now at 3..6 on direct page
ldy #4          ;offset to high word of ptrToPixImage
lda [3],y       ;got high word
tax             ;  in X
ldy #2          ;offset to low word of ptrToPixImage
lda [3],y       ;got low word
tay             ;  in Y
pld             ;restored direct page location
pla
pla             ;removed port pointer
```

  The `X` and `Y` registers now contain the base address of the screen.

- Do not assume things about being the frontmost window. Even if `FrontWindow` says you have the frontmost window, your `visRgn` may have pieces missing. For example, the title bar of your window may be partially under the menu bar. Or there may be a floating "windoid" (like Video Keyboard's window) over part of your window.

  For these reasons you should not draw directly to the screen without first examining your window's `visRgn`. Do not just check for rectangularity—your `visRgn` could be rectangular and parts of your window still be obscured. If you use QuickDraw for all your drawing, QuickDraw automatically clips drawing activity to be entirely within the `visRgn`, so this is not a problem.

- Don't access QuickDraw data directly; use QuickDraw routines instead. For example, to access SCB data, use the QuickDraw routines `GetSCB` and `SetSCB` instead of reading the hardware at $E19D00. CloseView may have those SCBs changed to reflect a magnified portion of the screen. Also use `GetColorEntry`, `SetColorEntry`, `GetColorTable`, and `SetColorTable`. Don't access the hardware directly.

- Try to allocate memory **after** starting the tools. If you want to allocate memory before starting tools, do not use special memory. (Set the `attrNoSpec` bit in the attributes.)

Further Reference

- *Apple IIGS Toolbox Reference*
- *Apple IIGS Firmware Reference*
- Apple II Video Overlay Card Development Kit (APDA)