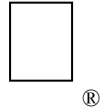


Apple II Technical Notes



Developer Technical Support

Apple IIGS

#81: Extended Control Ecstasy

Revised by:
1991

Dave Lyons July

Written by:
1990

C.K. Haun May

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      #oct1Data                  ; 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.