Macintosh
Technical Notes

Developer Technical Support

#306: Drawing Icons the System 7 Way

Revised by: Jim Mensch May 1992
Written by: Jim Mensch and David Collins October 1991

This Technical Note describes how to utilize the built-in System 7 icon drawing utility. Use this information to better conform to the System 7.0 visual human interface.

Introduction

With the introduction of System 7.0 for the Macintosh, Apple has defined a new look and feel for many screen elements that better utilize color. Among other elements, the icons drawn by the Finder and other system components have been redefined. While Apple has documented how to create this new look for most elements, Apple has not documented how to draw the icons the way the Finder does in System 7. This Technical Note discusses all the icon toolkits that the Finder uses to draw and manipulate the icons on the screen. Two of the calls, PlotIconID and PlotCIconHandle, are the ones you will probably use the most since they deal with simply drawing single icons to the screen. Other places in the Toolbox require that an icon family handle be passed to them to allow the drawing of color icons. The toolkit provides calls that allow you to create, draw, and manipulate these handles. What follows is a description of the new icon data structures and the calls in the icon toolkit.

The New 'ic' Type Resources

PlotIconID and PlotCIconHandle allow the use of standard CIcons as documented in *Inside Macintosh* Volume V and the use of a new set of icon resources utilized by the PlotIconID call. This new icon type is actually not a single resource but a collection of many different icons into

a family. Each member of the family has a consistent resource ID and a resource type indicating what type of icon data is stored in that particular resource. Currently Apple has defined three sizes of icons and three different bit depths for each size. The sizes are large (32x32 pixels), small (16x16 pixels), and mini (12x12 pixels), and the bit depths are 1, 4, and 8. The actual resource types are defined as follows:

```
Large1BitMask       =       'ICN#';
Large4BitData       =       'icl4';
Large8BitData       =       'icl8';
Small1BitMask       =       'ics#';
Small4BitData       =       'ics4';
Small8BitData       =       'ics8';
Mini1BitMask        =       'sicn';
Mini4BitData        =       'icm4';
Mini8BitData        =       'icm8';
```

The 1-bit-per-pixel member of each size also contains the mask data for all icons of that size (yes, this means that all your icons of a certain size must have the same mask). A 1-bit-per-pixel member

must exist for each icon size you want PlotIconID to use. The icon size to use is determined by the size of the destination rectangle. If the destination rect is greater than 16 pixels on a side then the large icon will be used. If the destination rect is 13–16 pixels on both sides the small icon will be used. If the destination is 12 or less on each side the mini icon will be used. The bit depth is determined by the device of the grafPort you plot into at drawing time. Be sure that you always create a color grafPort any time you want to use color icons.

Icon Families (or Suites and Caches As the Tool Set Refers to Them)

An icon family is simply a collection of icon handles that contain up to one image of each bit depth and size for a given icon. By using families, you remove the need to determine which size or depth of icon to use when drawing into a given rectangle. Several system routines can take an icon family handle when an icon is requested (the Notification Manager for example) so that the proper color icons can be used if available. An icon family can be fully populated (every possible size or depth available) or it can have only those icons that exist or are needed. In the case of a sparsely populated icon family, if the proper icon is not available the icon tool set will pick an appropriate substitute that will produce the best results.

An icon cache is a family that also has a ProcPtr and a refCon. The main difference between a cache and a family is that the elements of the cache's array are sparsely populated. When using an icon cache, the system either will use the entry in the icon family portion of the cache, or if the desired element is empty, it will call your procPtr and request the data for the icon. The Proc has the following calling convention:

```
Function GetAnIcon(theType:ResType; yourDataPtr:Ptr):Handle;
```

This function should return either the icon data to be used to draw or NIL to signify that this entry in the icon cache does not exist. Icon caches can be used with all icon family calls and have a few extra calls used to manipulate them.

Now that we know about the different data types let's examine how to manipulate the drawing:

Drawing Modes or Transforms

In addition to being drawn in various sizes and bit depths, icons can be drawn with different "Modes" or transforms. Transforms are analogous to certain Finder states for the icons. For example, the transform that you would use to show an icon of a disk that has been ejected is ttOffline. Here is a list of the current transforms that are available:

```
{ IconTransformType values }
ttNone                  =     $0;
ttDisabled              =     $1;
ttOffline               =     $2;
ttOpen                  =     $3;
ttSelected              =     $4000;
ttSelectedDisabled      =     (ttSelected + ttDisabled);
ttSelectedOffline       =     (ttSelected + ttOffline);
ttSelectedOpen          =     (ttSelected + ttOpen);
```

The actual appearance of the icon drawn by each transform type may vary with future system software, so you should always try to use the transform that best fits the state it represents in your application. In this way you will be consistent with any possible future changes to the look and feel of regular system icons. Note that the ttSelected transform can be added to any of the other

transform types. Additional transform types exist for displaying the icon of a file inside your application that use the Finder label colors to color the icon. To determine the proper label for a file's icon, you can check bits 1–3 of the fdFlags field in the file's Finder info (See the File Manager chapter in *Inside Macintosh* Volume IV for more information). These bits contain a number from 0 to 7. Simply add the corresponding ttLabel value to the transform that you give the call. The label values are defined like this:

```
ttLabel0                    =       $0000;
ttLabel1                    =       $0100;
ttLabel2                    =       $0200;
ttLabel3                    =       $0300;
ttLabel4                    =       $0400;
ttLabel5                    =       $0500;
ttLabel6                    =       $0600;
ttLabel7                    =       $0700;
```

Alignment

Most icons do not fully fill their rectangle, and it is sometimes necessary to draw an icon relative to other data (like menu text). In these instances it would be nice to be able to have the icon move in its rectangle so that it will be at a predictable location in the destination rectangle. Therefore, when drawing an icon you can pass one of these standard alignments in the alignment parameter or you can add a vertical alignment to a horizontal alignment to create a composite alignment value.

```
atNone                      =       $0;
atVerticalCenter            =       $1;
atTop                       =       $2;
atBottom                    =       $3;
atHorizontalCenter          =       $4;
atLeft                      =       $8;
atRight                     =       $C;
```

And Now (Drum Roll Please) the Calls and What to Pass

Now that we have defined every major data type we can think of, here is the real meat of this Tech Note: the actual calls themselves. I am providing only the Pascal and C interfaces here since they do not appear in your current MPW Interfaces folder. Disk copies of these can be found on AppleLink and on current and future versions of the *Developer CD Series* disc.

Icon Family Calls

```
Function NewIconSuite( var theSuite:Handle):OSErr;
```

This call returns an empty icon family handle with all members set to NIL.

```
Function AddIconToSuite( theIconData: Handle; theSuite:Handle;
                         theType:ResType):OSErr;
```

This call will add the data in theIconData into the suite at the location reserved for theType of icon data. This call will replace any old data in that slot without disposing of it, so you may want to call GetIconFromSuite to obtain the old handle (if any) to dispose. This call will be used most often with the NewIconSuite call to fill the empty family after it's created.

```
Function GetIconFromSuite( Var theIconData:Handle; theSuite:Handle;
                           theType:ResType):OSErr;
```

This call will return a handle to the pixel data of the family member of theSuite specified by theType. If you intend to dispose of this handle, be sure to call AddIconToSuite with a NIL handle to zero out the family entry.

```
Function ForEachIconDo(    theSuite: Handle; theSelector:Longint;
                           actionProc:procPtr; yourData: UNIV ptr):OSErr;
```

This routine will call your actionProc for each icon in the family specified by theSelector and theSuite. TheSelector is a bit level flag that specifies which family members to operate on; they can be added together to create composite selectors that work on several different family members. The values for theSelector are as follows:

```
{ IconSelectorValue masks }
svLarge1Bit                =      $00000001;
svLarge4Bit                =      $00000002;
svLarge8Bit                =      $00000004;
svSmall1Bit                =      $00000100;
svSmall4Bit                =      $00000200;
svSmall8Bit                =      $00000400;
svMini1Bit                 =      $00010000;
svMini4Bit                 =      $00020000;
svMini8Bit                 =      $00040000;
svAllLargeData             =      $000000ff;
svAllSmallData             =      $0000ff00;
svAllMiniData              =      $00ff0000;
svAll1BitData              =      (svLarge1Bit + svSmall1Bit + svMini1Bit);
svAll4BitData              =      (svLarge4Bit + svSmall4Bit + svMini4Bit);
svAll8BitData              =      (svLarge8Bit + svSmall8Bit + svMini8Bit);
svAllAvailableData         =      $ffffffff;
```

The action procedure that gets called for each icon type selected for the family is a Pascal type function with the following interface:

```
Function ActionProc(var theIconData:Handle; theType:ResType; yourDataPtr: UNIV ptr):OSErr;
```

theIconData is passed by reference here so that your routine can modify the contents of the suite directly. yourDataPtr is the value passed when you called forEachIconDo; it allows you to easily communicate with your application. The action procedure returns an OSErr; if any value other than noErr is returned, forEachIconDo will stop processing immediately and return the error passed. (Note: This implies that the icons selected may only be partially operated on.) There is no guaranteed order in which the icons get operated on.

```
Function GetIconSuite(   Var theSuite: Handle; theID:Integer;
                           theSelector:Longint):OSErr;
```

GetIconSuite will create a new icon family and then fill it with the icons with the passed ID, of the indicated types in theSelector, from the current resource chain. This is the call you will probably use most often to create an icon family. Note: If you SetResLoad(False) before making this call, the suite will be filled with unloaded resource handles.

```
Function PlotIconSuite(      theRect:Rect; alignment:Integer; transform:Integer;
                             theSuite:Handle):OSErr;
```

This call renders the proper icon image from the passed icon family based on the bit depth of the display you are using and the rectangle that you have passed. Alignment and transform are applied to the icon selected for drawing and then the icon is plotted into the current grafPort. PlotIconSuite

chooses the appropriate icon based primarily on size; once the proper icon size is determined (based on the destination rectangle) the present member of that size with the deepest bit depth that the current device can use is selected. A size category is considered present if the black and white member (with mask) is present, ICN#, ics#, or icm#. PlotIconSuite can be used for both picture accumulation and printing.

```
Function DisposeIconSuite( theSuite:Handle; disposeData:Boolean):OSErr;
```

This call disposes the icon family handle itself; in addition if disposeData is true, any of the icon data handles that do not belong to a resource fork will also be disposed.

```
Function SetSuiteLabel( theSuite: Handle; theLabel:Integer):OSErr;
```

This call allows you to specify a label that is used to draw an icon of this suite when noLabel is specified in PlotIconSuite. This is used primarily when you want to make sure that a family passed to a system routine gets drawn with the proper label. The default label can be overridden by specifying a label in PlotIconSuite.

```
Function GetSuiteLabel( theSuite:Handle; var theLabel:Integer):OSErr;
```

returns any label set with SetSuiteLabel previously.

Icon Cache Calls

In addition to the icon family calls above, icon caches have these additional calls:

```
Function MakeIconCache( VAR theCache:Handle; GetAnIcon:ProcPtr;
       yourDataPtr:Ptr):OSErr;
```

This call creates an empty icon cache similar to NewIconSuite, and associates the additional icon loading proc and data value with the family.

```
Function LoadIconCache(      theRect:Rect; alignment:Integer; transform:Integer;
                             theSuite:Handle):OSErr;
```

This call allows you to preflight the loading of certain elements of your icon cache. This is handy when you suspect that certain drawing operations will occur at a time not convenient for you to load your icon data (for example, when your resource fork might not be in open chain). LoadIconCache takes the same parameters as PlotIconSuite and uses the same criterion to select the icon to load. Be sure that the grafPort is set properly before you make this call since it is part of the criterion for determining which icon to load.

The following four calls are provided to allow you to change the dataPtr and procPtr associated with an icon cache:

```
Function GetCacheData( theCache:Handle; VAR yourDataPtr:Ptr):OSErr;
Function SetCacheData( theCache:Handle;  yourDataPtr:Ptr):OSErr;
Function GetCacheProc( theCache:Handle; VAR theProc:ProcPtr):OSErr;
Function SetCacheProc( theCache:Handle;  theProc:ProcPtr):OSErr;
```

Plotting Icons Not Part of a Suite

The following two calls are grouped because they are very similar. These routines let you simply plot an icon to the screen without having to create an icon suite. They are also good if you have a cicn instead of an icon family.

```
FUNCTION PlotIconID(      TheRect: Rect;
                    Align: Integer;
                    Transform: Integer;
                    TheResID: INTEGER): OSErr;

FUNCTION PlotCIconHandle(TheRect: Rect;
                    Align: Integer;
                    transform: Integer;
                    TheCIcon: CIconHandle): OSErr;
```

TheRect is the destination rectangle to draw the indicated icon into.

Align is the alignment method to use if the icon does not exactly fit the rectangle given. Pass zero for this value. See the next version of this Tech Note for more information on alignment.

Transform indicated the desired appearance of the icon on the screen.

TheResID is the resource ID of the family of 'ic' type resources to use. If the correct bit depth or size required is not defined, the closest-fitting one will be used.

TheCIcon is a handle that you get to a standard QuickDraw color icon. Call GetCIcon to load these and do not forget to dispose of it when you are done (sometimes they can take up quite a bit of memory).

Both functions return an error code if all did not go well with the drawing, or in the case of the PlotIconID call, if the indicated icon family could not be used.

Miscellaneous Calls

```
Function GetLabelColor(     labelNumber:Integer; var labelColor:RGBColor;
                            VAR LabelString:str255):OSErr;
```

This call returns the actual color and string used in the label menu of the Finder and the label's Control Panel. This information is provided in case you wish to include the label text or color when displaying a file's icon in your application.

```
Function IconSuiteToRgn(      theRgn: RgnHandle; iconRect:Rect
                              Alignment:Integer; iconSuite:Handle):OSErr;
Function IconIDToRgn(         theRgn: RgnHandle; iconRect:Rect
                              Alignment:Integer; iconID:Integer):OSErr;
```

These routines will create a region from the mask of the icon selected by the Rect and Alignment passed. This will allow you to do accurate hit testing and outline dragging of an icon in your application. TheRgn handle must have been previously allocated before you make this call.

```
Function RectInIconSuite(    testRect:Rect; iconRect:Rect; Alignment:Integer;
                             IconSuite:Handle):Boolean;
Function RectInIconID(       testRect:Rect; iconRect:Rect; Alignment:Integer;
                             IconID:Integer):Boolean;
Function PtInIconSuite(      testPoint:Point; iconRect:Rect; Alignment:Integer;
                             IconSuite:Handle):Boolean;
Function PtInIconID(         testPoint:Point; iconRect:Rect; Alignment:Integer;
                             IconID:Integer):Boolean;
```

These calls hit test the passed point or rect again the icon indicated. The iconRect, alignment, and grafPort should be the same as when the icon was drawn last. They return true if the point is in the icon mask, or if the rect intersects the icon mask.

Glue for C and Pascal

Since the standard interface files do not contain the glue for these calls, I am going to include it here since Tech Notes sometimes get distributed in electronic format and if all else fails you can copy and paste it.

```
{ Pascal Glue }
FUNCTION PlotIconID(theRect: Rect;align: Integer;transform: Integer;
       theResID: INTEGER): OSErr; INLINE $303C, $0500, $ABC9;
FUNCTION NewIconSuite(VAR theIconSuite: Handle): OSErr;     INLINE $303C, $0207, $ABC9;
FUNCTION AddIconToSuite(theIconData: Handle;theSuite: Handle;theType: ResType): OSErr;
       INLINE $303C, $0608, $ABC9;
FUNCTION GetIconFromSuite(VAR theIconData: Handle;theSuite: Handle;theType: ResType): OSErr;
       INLINE $303C, $0609, $ABC9;
FUNCTION ForEachIconDo(theSuite: Handle;selector: Integer;action: ProcPtr;
       yourDataPtr: Ptr): OSErr;  INLINE $303C, $060A, $ABC9;
FUNCTION GetIconSuite(VAR theIconSuite: Handle;theResID: INTEGER;
       selector: Integer): OSErr; INLINE $303C, $0501, $ABC9;
FUNCTION DisposeIconSuite(theIconSuite: Handle;disposeData: BOOLEAN): OSErr;
       INLINE $303C, $0302, $ABC9;
FUNCTION PlotIconSuite(theRect: Rect;align: Integer;transform: Integer;
       theIconSuite: Handle): OSErr;     INLINE $303C, $0603, $ABC9;
FUNCTION MakeIconCache(VAR theHandle: Handle;makeIcon: procPtr;
       yourDataPtr: UNIV Ptr): OSErr;    INLINE $303C, $0604, $ABC9;
FUNCTION LoadIconCache(theRect: Rect;align: Integer;transform: Integer;
       theIconCache: Handle): OSErr;     INLINE $303C, $0606, $ABC9;
FUNCTION GetLabel(labelNumber: INTEGER; VAR labelColor: RGBColor;
       VAR labelString: Str255): OSErr;  INLINE $303C, $050B, $ABC9;
FUNCTION PtInIconID(testPt: Point; iconRect: Rect; align: Integer;
        iconID: INTEGER): BOOLEAN;       INLINE $303C, $060D, $ABC9;
FUNCTION PtInIconSuite(testPt: Point; iconRect: Rect;align: Integer;
       theIconSuite: Handle): BOOLEAN;   INLINE $303C, $070E, $ABC9;
FUNCTION RectInIconID(testRect: Rect; iconRect: Rect;align: Integer;
       iconID: INTEGER): BOOLEAN; INLINE $303C, $0610, $ABC9;
FUNCTION RectInIconSuite(testRect: Rect; iconRect: Rect;align: Integer;
       theIconSuite: Handle): BOOLEAN;   INLINE $303C, $0711, $ABC9;
FUNCTION IconIDToRgn(theRgn: RgnHandle; iconRect: Rect;align: Integer;
       iconID: INTEGER): OSErr;    INLINE $303C, $0913, $ABC9;
FUNCTION IconSuiteToRgn(theRgn: RgnHandle; iconRect: Rect;align: Integer;
       theIconSuite: Handle): OSErr;     INLINE $303C, $0914, $ABC9;
FUNCTION SetSuiteLabel(theSuite: Handle; theLabel: INTEGER): OSErr;
       INLINE $303C, $0316, $ABC9;
FUNCTION GetSuiteLabel(theSuite: Handle): INTEGER; INLINE $303C, $0217, $ABC9;
FUNCTION GetIconCacheData(theCache: Handle; VAR theData: Ptr): OSErr;
       INLINE $303C, $0419, $ABC9;
FUNCTION SetIconCacheData(theCache: Handle; theData: Ptr): OSErr;
       INLINE $303C, $041A, $ABC9;
FUNCTION GetIconCacheProc(theCache: Handle; VAR theProc: ProcPtr): OSErr;
       INLINE $303C, $041B, $ABC9;
FUNCTION SetIconCacheProc(theCache: Handle; theProc: procPtr): OSErr;
       INLINE $303C, $041C, $ABC9;
FUNCTION PlotCIconHandle(theRect: Rect; align: INTEGER; transform: INTEGER;
       theCIcon: CIconHandle): OSErr;    INLINE $303C, $061F, $ABC9;

/* C Glue */
pascal OSErr PlotIconID(const Rect *theRect, short align, short  transform, short theResID)
        = {0x303C, 0x0500, 0xABC9};
pascal OSErr NewIconSuite(Handle *theIconSuite) = {0x303C, 0x0207, 0xABC9};
pascal OSErr AddIconToSuite(Handle theIconData,Handle theSuite,ResType theType)= {0x303C, 0x0608, 0xABC9};
pascal OSErr GetIconFromSuite(Handle *theIconData,Handle theSuite,ResType theType)= {0x303C, 0x0609, 0xABC9};
pascal OSErr ForEachIconDo(Handle theSuite,short selector,ProcPtr action,void *yourDataPtr)
        = {0x303C, 0x080A, 0xABC9};
pascal OSErr GetIconSuite(Handle *theIconSuite,short theResID,short selector)= {0x303C, 0x0501, 0xABC9};
pascal OSErr DisposeIconSuite(Handle theIconSuite,Boolean disposeData)= {0x303C, 0x0302, 0xABC9};
```

```
pascal OSErr PlotIconSuite(const Rect *theRect,short align,short transform,Handle theIconSuite)
        = {0x303C, 0x0603, 0xABC9};
pascal OSErr MakeIconCache(Handle *theHandle,ProcPtr makeIcon,void *yourDataPtr)= {0x303C, 0x0604, 0xABC9};
pascal OSErr LoadIconCache(const Rect *theRect,short align,short transform,Handle theIconCache)
        = {0x303C, 0x0606, 0xABC9};
pascal OSErr GetLabel(short labelNumber,RGBColor *labelColor,Str255 labelString)= {0x303c, 0x050B, 0xABC9};
pascal Boolean PtInIconID(Point testPt,Rect *iconRect,short alignment,short iconID)= {0x303c, 0x060D, 0xABC9};
pascal Boolean PtInIconSuite(Point testPt,Rect *iconRect,short alignment,Handle theIconSuite)
        = {0x303c, 0x070E, 0xABC9};
pascal Boolean RectInIconID(Rect *testRect,Rect *iconRect,short alignment,short iconID)
        = {0x303c, 0x0610, 0xABC9};
pascal Boolean RectInIconSuite(Rect *testRect,Rect *iconRect,short alignment,Handle theIconSuite)
        = {0x303c, 0x0711, 0xABC9};
pascal OSErr IconIDToRgn(RgnHandle theRgn,Rect *iconRect,short alignment,short iconID)
        = {0x303c, 0x0613, 0xABC9};
pascal OSErr IconSuiteToRgn(RgnHandle theRgn,Rect *iconRect,short alignment,Handle theIconSuite)
        = {0x303c, 0x0714, 0xABC9};
pascal OSErr SetSuiteLabel(Handle theSuite, short theLabel)= {0x303C, 0x0316, 0xABC9};
pascal short GetSuiteLabel(Handle theSuite)= {0x303C, 0x0217, 0xABC9};
pascal OSErr GetIconCacheData(Handle theCache, void **theData)= {0x303C, 0x0419, 0xABC9};
pascal OSErr SetIconCacheData(Handle theCache, void *theData)= {0x303C, 0x041A, 0xABC9};
pascal OSErr GetIconCacheProc(Handle theCache, ProcPtr *theProc)= {0x303C, 0x041B, 0xABC9};
pascal OSErr SetIconCacheProc(Handle theCache, ProcPtr theProc)= {0x303C, 0x041C, 0xABC9};
pascal OSErr PlotSICNHandle(const Rect *theRect,short align,short transform,Handle theSICN)
        = {0x303C, 0x061E, 0xABC9};
pascal OSErr PlotCIconHandle(const Rect *theRect,short align,short transform,CIconHandle theCIcon)
        = {0x303C, 0x061F, 0xABC9};
pascal OSErr SetLabel(short labelNumber, const RGBColor *, ConstStr255Param)
        = {0x303C, 0x050C, 0xABC9};
```

Further Reference:

- *Inside Macintosh*, Volume V, QuickDraw chapter