Cognits: Writing Portable Code

Steven M. Lewis, Ph.D.
Member of the Technical Staff
Computer Systems Division
The Aerospace Corporation, El Segundo, Calif. 90245
slewis@aero.org

Abstract

Writing good, user friendly, graphical applications on the Macintosh is difficult . Even more challenging is writing a single application capable of running across a number of GUIs. This paper is a case study of the Cognits library, a portable class library that runs on the Macintosh, X Windows and MS Windows . The discussion will specifically concentrate on writing code that will port between the Macintosh and MS Windows. Three areas will be addressed: the general principles in the design of portable systems.; corresponding features of the Mac and Windows operating systems, especially drawing and interaction with the user.;and unification of higher level functionality. The object of a portable library is to allow a single collection of source code to compile into applications that will run under multiple systems. While higher level code may manipulate system dependent structures, well designed portable systems should not require changes in the text of code to generate similar results. on multiple systems Cognits divides code into a system dependent layer and a larger system independent layer. The system dependent interacts with the underlying GUI and defines a ''virtual machine' used by high level, system independent code. The most important step in designing portable code is to unify disparate conceptual views of elements of the applications into a single uniform framework. Correspondences must be made between elements of the two systems leading to a single integrated approach to drawing, event handing and controls. Important decisions must be made concerning implementation of controls, whether to use native or portable look and feel, and where in an application events are handled. Finally, there are differences in the function of advanced features such as standards dialogs and interprocess communication. This paper considers a number of choices in implementing Cognits and presents code to implement many of the common Macintosh drawing commands under Windows.

Introduction

The capabilities that the Macintosh pioneered in 1984 are no longer unique. X Windows (coupled with several overlying window managers), NeXTStep, and Microsoft Windows offer capabilities similar to those on the Macintosh. Increasingly, we are called upon to deliver applications that will run on multiple platforms. This paper examines the problems in writing code that is portable across several platforms, especially between Microsoft Windows and the Macintosh.

Portability is a major design driver. It is difficult to port an application that has not been designed from the outset for portability. The requirements of portability must continually be in the mind of the developer during the development of the application. Portable applications always represent a compromise. It may not be possible to take full advantage of the features of any one system if those features are not offered or cannot be replicated on other

systems. A portable application tends to be written to the lowest common denominator. When features are not available on one system, the developer has three choices: Decline to use those features, write code to implement an appropriate facsimile or write applications which, while using the features when present, will perform adequately in their absence, reducing its functionality to accommodate available capabilities.

This paper examines the choices I made in developing a portable class library and the approach I took to allow the same code to run on both Microsoft Windows and the Macintosh. My general approach was to define a portable API (Application Program Interface) that could be implemented on top of any target system. Higher level code makes calls to this API without knowledge of the underlying system. The proper use of this approach requires significant compromises but produces capable, portable applications.

General Considerations in Writing Portable Code

Several principles are involved in writing portable code. First, divide the code into two sections. The first portion is the code that is aware of which system under which it is running. This code must necessarily be different for each different machine. It is the only section supporting direct calls to the underlying GUI. The second portion of the code is system independent. This code may include macros and structures that can vary from system to system. The same text (with system dependent definitions) must operate properly on all systems. A major objective is to minimize the size of the system dependent code and maximize the size of the code that can run under any system. At each point the developer is faced with decisions that determine whether sections of code go in the system dependent or the system independent sections. Any code in the system dependent portion must be replicated, validated and maintained at least twice to assure not only proper functionality but also equivalent behavior on all systems.

System independent code considers operations at a conceptual level rather than an implementation level. Functionality is presented in high level concepts rather than the low level details to implement them. Consider a few common operations. The display of a popup menu consists of a request to the user to choose among a collection of names with the request appearing at a location on the screen and the return being either the name chosen or NULL if no choice is selected. One may write
*const char *ChoosePopUP(*
*const char **Choices,*
short NChoices,
short DefaultChoice,
Point Location);

This command can easily be written on the Macintosh to build a popup menu from an array of choices, display it , get the resultant choice, destroy the menu, and return the resultant choice. The same function can easily be written under Windows. While the internal data types used in the two implementations will be quite different, the end result will be the same. When this is the only way high level code can use pop up menus, then that code will be system independent.

Another example is file opening. All systems support file manipulation with the standard UNIX commands: fopen, fclose, fread, fwrite. Here files are opened with a string representing the path and manipulated with a file pointer. A portable approach to file manipulation would offer functions such as shown below which return the name of an existing file to pass to fopen:

*char *GetExistingFileName(*
*const char *DirectoryName,*
*const char *DefaultFileName,*
*const char *Prompt,*
Point Location)

Once again, SFGetFile can easily be used to generate this call, translating the

selection into a full path name. The same call can be implemented under Windows using the standard windows dialogs. More sophisticated calls can add filtering in a system independent manner.

Note that all calls use portable data types such as strings, Points, Rects and UNIX style FILEs. In almost all cases, the developers real intentions can be expressed in these portable constructs. System dependent constructs such as FSSpec or MenuHandles are merely low level means of implementing a request. It is not necessary for higher level code to deal with these constructs . Minimizing the fraction of the code that needs to work at this level enhances the portability and maintainability of the resultant code. In my work, the system independent code does not even include any system dependent header files. Standard C header files are available on all systems and these are included. Direct access to system traps or any fields in system dependent structures such as GrafPorts is not allowed. These practices give the code a very generic look and feel.

The next general principle is to '"Just Say No". An application that seeks to be portable must offer similar services on all target systems. If the application uses a service such as the ListManager, WorldScript or AppleEvents on one system, it must usually find a way to offer similar services on all systems. If similar capabilities are not available on other systems, the developer must code them from more primitive operations. At the end of this process, the developer has been required to develop all the code to perform the service in order to satisfy the needs of the most limited system. Once the code is running well there is a strong argument to use the developed code on all systems since this moves the code from the system dependent to the system independent portion of the application. In the current version of Cognits, TextFields are implemented by using more primative calls for greater control and portability. (see below). The utilization of features not available on all target systems should be avoided, especially in critical sections of the application.

A third principle is to generalize function.

Interprocess communication is available on all systems. DDE's (A Windows specific interprocess communication scheme) and AppleEvents are not. In developing portable code, the developer needs to create a common model that can utilize or create features common to both modes of interprocess communication. It is not necessary for a portable application to use all available features even within AppleEvents or DDE's. It is, however, necessary that a rich enough feature set be offered to allow applications to achieve needed functionality.

Fourth, some compromises must be accepted. There is no easy way to draw text under Windows with a pattern pen. A file's extension may be used to represent data type under Windows but there is no easy way to represent the file's creator. On the other hand, Pulldown Menus under Windows may be displayed and manipulated with the keyboard while this feature is not available on the Macintosh. Applications must be 'aware' of limitations of portability and be prepared to work within these bounds. This will involve minimizing the use of non-portable features, accepting some unavoidable differences and making sure these do not affect the core functionality.

One important issue in building portable applications is the issue of look and feel. If the same application is being delivered on several systems, the developer has three options for the application's look and feel. He can support a common look and feel. He can use the look and feel of each native platform. Supporting native look and feel can add to the complexity of a portable design by affecting the operation of higher level, system independent code. With significant additional effort, it is possible can allow the user to select the application's look and feel.

Contrast Between Windows and the Macintosh

Memory

Both the Macintosh and Windows offer handles to allow the allocation of relocatable blocks of memory. In both cases, system dependent structures are frequently handles to the actual data. Handles are relics of the days when both the Mac and the PC had very limited memory. Today, both systems offer virtual memory that, together with the general availability of cheaper memory, massively increases the memory available to an application. On both the Mac and the PC, advanced C compilers implement malloc in manner that reduces memory fragmentation. The use of malloc for program structures is portable and rarely costly. Since, as I discuss below, all system dependent structures must be treated as opaque by system independent portions of the system, the system independent code can ignore all uses of handles. All accessible memory can be treated as pointers.

Classically the PC has had to support a number of memory models that mix 16 and 32 bit pointers. This forced the distinction between far and near pointers. At this time there is little reason not to use the large model under Windows 3.1 that treats all pointers as 32 bits (segment + offset). Under Windows NT a flat 32 bit address space is available. Windows defines a number of data types such as LPSTR and LPRECT. Under the interesting memory models these usually translate into more conventional types: char * and Rect *.

Drawing

Ports and Bitmaps

All GUIs implement three basic structures. There is some structure into which the system can draw. On the Macintosh this is a GrafPort (CGrafPort and GrafPort will be synonymous for this discussion.) . In Windows, the equivalent structure (with significant differences discussed below) the structure is an HDC. In the interests of neutrality I use a system dependent type: DrawPort to represent both structures. Each has ways of specifying font, color, pen type and clipping.

Associated with every DrawPort is a region for drawing that I call a BitMapPtr. This region may be a window on the screen or an off screen buffer. Under Windows it is possible to change the BitMapPtr associated with a DrawPort. On the Macintosh, while this is possible, usually the two are tightly associated. On the Mac, the DrawPort has foreground and background colors and pattern built in. On the PC an HDC has a fill and line brushes holding this information. This difference requires additional information to be attached to the DrawPort structure. To emulate the behavior of the Macintosh when the pen pattern is changed, the PC must build new brushes using the current foreground and background colors. This information therefore must be part of the DrawPort. The Structure I created for a DrawPort has an HDC, foreground and background Colors, pattern, font and an associated window.

A third GUI object is a window on the screen. Every window is associated with a DrawPort and implicitly contains a BitMapPtr. Neither Mac nor PC encourages the direct manipulation of a window's bitmap. In MS Windows and many X Windows systems each control is treated as a separate child window. Child windows must be located within a parent window and are considered as dependent on the parent. For portability, the windows referred to in this paper are only parent windows.

The Macintosh supports the concept of a current Port. Drawing is performed by calling SetPort (or SetGworld) and then performing drawing. Under MS Windows (and also X Windows) the affected DrawPort is part of all drawing calls. A simple approach is to provide Windows with a hidden global DrawPort that can be set with calls to SetPort and returned by

GetPort. Rather than using the same name as the Macintosh calls, I use similar names: SetThePort and GetThePort.

The Mac and Windows support the concept of pulldown and popup menus. On the Macintosh, pulldown menus are located in a global menu bar representing the menus of the active application. In MS Windows, menu bars are located within individual windows with no global menu bar. The appendix lists neutral routines for building and manipulating menus.

Resources

Resources on the Macintosh have the following characteristics. Resources are identified by a four letter type a resource number and an optional name. Resources may be read, written, and searched by either name or number. They may be part of an application, a separate resource file or an other file's resource fork. In Windows resource name and type are both strings. Resources are developed at compile time and may not be modified by a running application.

The unification provided to deal with resources is to minimize their use. Resources dealing with non portable types such as Dialog, control and code are avoided. String, Picture and Icon resources are treated as read only. Icon and Picture resources return appropriate opaque types. Cognits resource manager allows all applications to write custom resources. On the Macintosh these custom resources are treated as conventional resources. On the PC, a separate file similar to a preference file is created in the system directory to hold an application's read/write resources.

Opaque Objects		
Opaque	Mac	Windows
DrawPort	CGrafPort	HDC
WINDOWPtr	Window	HWND+
MENUBARPtr	MenuBarHandle	Custom
MENUPtr	MenuHandle	HMENU
MENUItem	Custom	Custom
ICONPtr	CIcon	
PICTUREPtr	PictureHandle	MetaFile
DIRECTORY	int	string
REGIONPtr	RgnHandle	HRGN

Implementation of Macintosh Drawing Calls

The portable interface uses standard Macintosh drawing calls for most drawing operations. This section illustrates samples of how such calls are implemented under Windows. (I will include complete code listings for the basic calls in the conference CD.) The commands I support are Paint, Frame, Fill, Invert, PointIn and Designate (see below) for structures Rect, Oval, RoundRect, Arc and Polygon. All except Designate and PointIn are supported directly with standard Mac traps, so this section will be concerned with the generation of equivalent behavior on the PC.

The concept of a Macintosh pen is divided into three objects under Windows. A Brush is for filling areas. A Pen is for drawing lines and framing and TextColor is for drawing Text.

The system on the PC holds the following globals:

HDC CurrentDC - This is the equivalent of the current port on the Mac and is altered with a call to SetThePort.

refcolor TheForeColor - foreground color in Windows internal form.

refcolor TheBackColor - background color in Windows internal form.

boolean ValidCurrentBrush, ValidCurrentPen Setting the foreground color requires the construction of a number of brushes and pens. These are not actually built until a drawing command is issued. This flag is set true if the current set of pens and brushes does not require updating. Altering the foreground or background color will invalidate this flag, forcing new pens and brushes to be constructed before drawing.

ToLPRECT is a macro of mine that converts a Rect * to a Windows RECT.

Helper Routines

These routines are needed to build the needed Windows pens and brushes to implement the Windows equivalent of the standard Macintosh drawing calls.

```
void ValidateCurrentBrush(void)
{
 HBRUSH NewBrush,DisplacedBrush;

  if(ValidCurrentBrush) return;
   // Brush is Valid

   // Make a brush
    NewBrush =
MakePatternBrush(CurrentPattern);
  // Instantiate it
   DisplacedBrush =
SelectObject(CurrentDC,NewBrush);
  // Destroy old brush
   if(DisplacedBrush)
     DeleteObject((HANDLE)OldBrush);
    CurrentBrush = NewBrush;

  ValidCurrentBrush = true;
   // Now brush is valid
   // Make sure Pen is valid as well
   ValidateCurrentPen();
}

// Make a Pen with the current
// foreground color. Only solid
// pens allowed
void ValidatePen(void)
{
  HPEN OldPen = CurrentPen;
  HPEN NewPen,DisplacedPen;
  if(ValidCurrentPen) return;

  // make a new pen
  NewPen  =     CreatePen(
       0,
       CurrentLineWidth,
       TheForegroundColor);
  // select it
  DisplacedPen =
SelectObject(CurrentDC,NewPen);
// Dispose the Old pen
     DeleteObject(DisplacedPen);
// remember the New pen
  CurrentPen = NewPen;
  // Valid pen made
  ValidCurrentPen = true;
}
```

// This makes and Selects a brush from // a

Pattern and the current foreground
// and background colors
// return is the previous brush that // must be restored after a Fill.

```
HBRUSH  MakePatternBrush(const  Pattern *ThePattern)
{
 // Select this as the current brush
      OldBrush                   =
SelectObject(CurrentDC,NewBrush);
      return(OldBrush); // remember Old
so we can restore
}

// This selects as current and destroys
//the current brush
void SetAndDestroyBrush(
       HBRUSH OldBrush)
{
       HBRUSH PrevBrush  =
       SelectObject(CurrentDC,
           OldBrush );
       DeleteObject(PrevBrush  );
}

//
// This  code implements MakePatternBrush
// the critical routine to implement
// FillRect, FillOval under windows
//
// This is a BITMAPINFO object for a 2 Color bitmap
typedef struct {
  BITMAPINFOHEADER hd;
  RGBQUAD   bmiColors[16];
  char   d[32];    // actual data
} PatternDIData;


//  PURPOSE    : Make a brush with 2 colors
//   ForeColor on 1 and BackColor  on  0
*
HBRUSH MakePatternBrush(const char *ThePattern)
{
  PatternDIData  P;
  HBRUSH  TheBrush;
// treat Solid patterns as special cases
//  BLACK_PATTERN is all 1 s
```

```
  if(EquivalentPattern(ThePattern,
    BLACK_PATTERN)) {
// CreateSolidBrush is a Windows call
    TheBrush =
    CreateSolidBrush(TheForegroundColor);
    return(TheBrush);
  }

// Make a DeNovo Brush
// Clear the structure
  memset(&P,0,sizeof(PatternDIData));
// Standard initlialization
  SetupPatternInfo(&P,ThePattern);
//  Set up brush
  TheBrush =
`   CreateColoredPatternBrush(
    &P,
    &CurrentForeColor,
    &CurrentBackColor);
  return(TheBrush);
}
//
//  Initialize a  PatternDIData to a Pattern
//
void SetupPatternInfo(PatternDIData *P,
`     const char *ThePattern)
{
  P->hd.biSize =
sizeof(BITMAPINFOHEADER);
  P->hd.biWidth = 8;     // 8 by 8 bitmap
  P->hd.biHeight = 8;
  // Compression - none
  P->hd.biCompression = BI_RGB;
  P->hd.biPlanes = 1; // Always
  P->hd.biBitCount = 4; // ???
  // Map to nibbles
  Pattern4ToData(ThePattern,P->d);
  P->hd.biSizeImage =
    P->hd.biWidth * P->hd.biHeight *
    P->hd.biBitCount / 8 ;

}

//   Turn a pattern into a 32 char array
//  of packed 4 bits each nibble is 0 or 1

void Pattern4ToData(const char
*ThePattern,
  char *TheData)
{
  int i,k,j;
  unsigned int xh;
  j = 0;
  // bit pattern as data read out the
  for(i = 0; i < 8; i++) {
    xh = ThePattern[i];
    for(k = 0; k < 4; k++) {
      // Set nibble 1
      if(xh & 1)
        heData[j] = 0x10;
      else
        TheData[j] = 0;
      xh >>= 1;

        // set nibbble
      if(xh & 1)
        TheData[j] += 0x01;

        j++;
      xh >>= 1;
    }
  }
}

//
//  Make a Brush with a PatternDIData
//    structure setting the
//    colors to the Current Fore and
//   background colors
static HBRUSH CreateColoredPatternBrush(
  PatternDIData *P,
  const COLOR *ForeColor,
  const COLOR *BackColor)
{
  HBITMAP NewBitmap;
  HBRUSH  TheBrush;
  // Set the Colors
  P->bmiColors[0].rgbRed
`      = ForeColor->red;
  P->bmiColors[0].rgbBlue
    = ForeColor->blue;
  P->bmiColors[0].rgbGreen
    = ForeColor->green;

  P->bmiColors[1].rgbRed
    = BackColor->red;
  P->bmiColors[1].rgbBlue
    = BackColor->blue;
  P->bmiColors[1].rgbGreen
    = BackColor->green;

// Windows Call  CreateDIBitmap
  NewBitmap =CreateDIBitmap(CurrentDC,
```

```
    (BITMAPINFOHEADER *)P,
    CBM_INIT,      // Is Initialized
    (unsigned char *)P->d,
    (BITMAPINFO *)P,
    DIB_RGB_COLORS);
// Windows Call  CreatePatternBrush
  TheBrush =
    CreatePatternBrush(NewBitmap);
.// Destroy the Bitmap we Created
  DeleteObject(NewBitmap);
   return(TheBrush);
}
```

Implementation of Mac Drawing Calls

```
//
//  Windows version of FrameOval
//
void mswFrameOval(const Rect *TheRect)
{
   // NULL_BRUSH says do not fill
   HBRUSH OldBrush =
        SelectObject(CurrentDC,
        GetStockObject(NULL_BRUSH));

   ValidateCurrentPen();
        // make sure Pen is OK
   Ellipse(CurrentDC,
        TheRect->left,
        TheRect->top,
        TheRect->right,
        TheRect->bottom);
   // Restore original brush
   SelectObject(CurrentDC,OldBrush);
}
```

```
//
//  Windows version of FillRect
//
void mswFillRect(
        const Rect *TheRect,
        const Pattern *ThePattern)
{
    // convert to windows rect
   RECT WinRect = ToLPRECT(TheRect);
   // Create a brush from the pattern
        HBRUSH TheBrush =
        MakePatternBrush(
              *ThePattern);

        // Fill with the created brush
        FillRect(CurrentDC,&WinRect,
              TheBrush);
        // Destroy the brush
   DeleteObject(TheBrush);
}
```

```
// System Independent Polygon Structure
typedef struct {
 int NPoints;   // the number of Points
 Point *ThePoints;   // an array of
//vertices Point is the Mac Point
} POLYGON;

//
//  Windows version of InvertPoly
//
void mswInvertPoly(POLYGON *ThePolygon)
{
        HRGN TheRegion;
// Conversion is needed since Windows
// and WindowsNT
// use different POINT structures and
// POLYGON is system independent
LPPOINT WinPoints =
        PolyToWinPoints(ThePolygon);

        TheRegion =
          CreatePolygonRgn(WinPoints,
              ThePolygon->NPoints,
              WINDING);

        InvertRgn(CurrentDC,
              TheRegion);

        delete [] WinPoints;
        // get rid of the points
        DeleteObject(TheRegion);

}
```

Designate

Even on a single system it is difficult to specify screen depth and color capabilities . In writing code for multiple systems this becomes doubly difficult. In many applications the specific fill of an area is less important than the fact that one area is different from another. Consider, for example, the problem of drawing a pie chart. Each slice of the pie needs to be distinctive. There is, however, no reason to select a specific color for one

particular slice as long as the slice and labels identifying that slice are filled in the same manner. Designate commands, such as DesignateRect and DesignatePoly, take an index and ask the system to fill the area in a manner that makes areas with different indices appear different. Thus the code

```
DesignateRect(R1,1);
DesignateRect(R2,2);
DesignateOval(R3,1);
```

will draw a Rect filled with pattern1, a Rect filled with pattern 2 and an oval filled with pattern1. In one bit, monochrome systems, the Rects are filled with differing patterns. On color systems, the Rects are filled with solid colors for lower indices and different colored patterns for higher indices .

The color capability of the system may be temporarily disabled, for example when printing to a monochrome printer. Under these conditions fills with the Designate command will remain distinct although they will not replicate what the user sees on the screen.

Files

The portability Files may be addressed using UNIX file commands. These are implemented on all systems. However, there are a number of differences between files on the Mac and the PC. Files on the Mac may have 31 character names including spaces and other non alphanumeric characters. Case is preserved in file names but not used in searches. Directories in a full path name are separated by ':' and volumes by '::' . On the PC, file names have up to 8 characters of name plus an optional 3-character extension. Case is not remembered in file names. Directories in a full path name are separated by '\' and volumes by ':' . In addition to file name, files on the Macintosh store finder information, most importantly File and Creator types. The extension is the only data available in Windows to store this information. Windows NT will implement yet another file system supporting longer file names.

In the discussion above a system independent call for invoking a file open dialog is discussed.

Events

While the actual events supported on the Mac and Windows are very similar, the handling of those events is very different. is very different between Windows and the Macintosh. The normal Mac event handler is written

```
EventRecord TheEvent;

    WaitNextEvent(everyEvent,
        &TheEvent,10L,NULL);
    ... Code to Process Event
```

Under Windows the code to process a single event looks like this

```
    MSG msg;

// receive a message
    GetMessage(&msg, NULL, 0, 0); //
obscure, poorly documented
// processing
    TranslateMessage(&msg );
// send to the window's message handler
    DispatchMessage(&msg );
```

Every Window supplies a message handler routine. Events are passed directly to the Window's event handling routine and no further processing is expected. This represents a very different paradigm from the Mac with event handling being window based rather than global.

The Windows event handling procedure looks like the following code. TheWnd is the receiving window, Message is a message type, similar to Event.what field. wParam and lParam are two parameters whose interpretation depends on Message. For mouse events lParam holds vertical and horizontal mouse position in the upper and lower 16 bits of the parameter.

```
LRESULT CALLBACK windproc (
    HWND TheWnd,
    unsigned int Message,
     unsigned int wParam,
```

```
    LONG lParam)
{

    switch(Message)
    {
    case WM_COMMAND:
        ... Code ...
    case WM_LBUTTONDOWN:
        ... Code ...
        break;
    ... Many Other Cases ...
    // Pass unhandled messages
    // to Windows for processing
     default:
     return DefWindowProc(
       hAWnd, Message,
        wParam,lParam);
     }
}
```

Another major difference is that Windows supplies default event handlers. Events that a window cannot handle are passed to a default handler. This is not a bad approach. There is really no reason why dragging a window to a new location cannot be handled by the system with the application merely receiving an event indicating that a drag has occurred.

A significant difficulty with this design is that most of the application lies in the case statement associated with the window. This makes the code difficult to maintain and modularize. Cognits chose to modify Windows event handling to emulate the way events are handled on the Mac. The windproc routine was written so that the only action of each clause in the case statement was to determine if the event required handling by the application. Events requiring handling merely build a Macintosh style event record , set a flag showing that there is an event to handle and return. Events that can be handled by the system reset the flag so no event is passed on and default to the built in handler.

There are differences in the types of events available on the two systems. Windows supports a two or three button mouse whereas on the Mac multiple buttons are emulated by holding down keys while clicking the mouse. Keystrokes that are to be handled by the system, such as command keys, differ as well. In the interests of portability, Cognits preprocesses events dividing them into added types depending on modifiers. Mouse events, for example, are treated as if there are a large number of buttons on the mouse. Buttons are given neutral color names: Black, Red, Green ... so events are translated to BlackMouseDown, BlueMouseDown ... There is a system dependent mapping of buttons and keys to events so BlackMouseDown is pressing the most common button. RedMouseDown is Command Click on the Mac and both alt + LeftMouseDown and RightMouseDown on the PC. Double clicks are also mapped to colored buttons.

This preprocessing means that subsequent event handlers need not be concerned with testing keys to see what actions are required. A separate handler is generated for each type of mouse click. The one concession is that a system dependent call called GuaranteeMouseClick is provided. This should be called when a commitment is about to be made. It is used to make sure that the click being processed is not the first click in a double click sequence.

Standard events, updates, key strokes and mouse clicks, are quite similar across systems. Higher level events such as AppleEvents on the Mac or DDE events on the PC require other layers to process in a similar manner. These are discussed in detail below.

A number of events such as dragging a window, clicking a close box or resizing a window can easily be handled in centralized, generic code. MS Windows and X both operate in this manner with the window subsequently receiving notification of any relocation, resize or destruction. Cognits chooses to treat these and several others as 'SystemEvents'. These are filtered by the event handler and not passed to higher level code. AppleEvents also fall into this category since they are not handled by the normal event handling mechanism.

COLOR

Controls

Colors on the Macintosh are designated as RGBColor with 3 16 bit values representing red, green and blue. Windows uses several structures to represent color but in all cases a single byte is used for red, green, and blue values. Windows routines use a 32 bit COLORREF structure, generated using one of several possible macros from red, green, and blue data. When filling areas, Windows will by default fill the exact color requested by dithering palette colors. The effect is rarely pleasing. Under Windows Cognits forces color selection from the palette to suppress dithering. Cognits uses a Color structure as follows:

typedef struct {
        unsigned char red,green,blue;
        unsigned char palette;
} COLOR;

Here *palette* is an optional entry into a 256 color palette. This entry and its use is system and hardware dependent. Translation between this structure and the colors used by native calls is straightforward. Other calls allow systems with 256 color hardware palettes to support palette animation.

In the Macintosh, controls are regions of a window capable of responding to user events. MS Windows and X Windows treat controls as separate sub-windows capable of independently receiving and responding to events. Treating controls as separate windows pushes much of the work of clipping drawing and distributing events onto the operating system. There are three costs associated with this approach. First, sub-windows are much more expensive structures in terms of both memory and time than drawn regions within a window. Second, directly passing events to controls makes it more difficult for an application to control event handling. When all events are sent to a central event manager, a window, and finally an active control, there a number of places where events can be monitored, delegated, and intercepted. In the Windows model, where events are sent directly to the affected control, it is harder to control and monitor their flow. Third, the difference in the two models of control increases the problems of portability.

From the point of view of the operating system, a Cognits window is merely a blank canvas. All controls are active regions recognized by the application, not by the operating system. This is essentially the approach taken by Hypercard. It allows the location of controls to be determined at run time and to vary with the contents of the window. An extremely common Cognits paradigm is to create a window, add a number of controls, request that they distribute themselves with a reasonable algorithm, and finally resize window to the size needed hold the included controls.

Buttons, Radio Buttons, CheckBoxes and scroll bars are not difficult to implement with elementary drawing commands. For Radio Buttons and CheckBoxes there are real advantages to this approach since groups of these may be considered as a single control and drawn and managed accordingly. This allows the action of a RadioButton to be treated as part of the

control's handler rather than as part of the application's. It also allows groups of CheckBoxes to be sized so that all elements of the group have similar size.

Text Fields

Both the Macintosh and Windows offer the equivalent of a TEHandle. In earlier implementations of a Text Control on the Macintosh, a real TEHandle was embedded into a higher level object. Later versions implemented the equivalent functionality using basic text draw commands. The implementation is based on an excellent set of articles by Martin Minnow (Minnow 90a,b) The decision to reimplement as complex a structure as a TEHandle was not taken lightly. Portability, the ability to deliver the same functionality on all systems was a major factor in this decision. While MS Windows supplies a text control similar to a TEHandle, its operation at both user and code level is not identical with the Macintosh control. X Windows supplies no such control and forces a developer to write the needed code anyway.

Coded text fields offer both advantages and disadvantages over the standard controls. Cognits text fields do not offer styled text or mixing fonts, sizes, and styles within a single field. This capability was not felt to be important in most applications. Cognits does support editing text exceeding 32K. Once text fields were coded, Cognits distinguished a number of classes. Output-only fields, which support display but not editing, can be considerably simpler than fields requiring full, multiline editing capabilities. Output-only fields can easily be subclassed to support tabs, multicolor text, and column delimiters - capabilities that are difficult to mix with the ability to edit the text. The vast majority of editable text fields support only a single line and can have a considerably simpler structure than multiline fields. Event handling in multiline fields is modified so that, in addition to double clicks supporting selection by word, triple clicks support selection by line, and quadruple clicks select the entire field. These varied capabilities show that while there is a significant investment in emulating system-supplied capabilities, there are rewards in greater functionality and control.

Higher-Level Operations

Both Windows and the Macintosh provide dialogs for common high level operations including, opening files, selecting directories, and selecting colors. A portable program should be able to use these services. The key in all cases is to convert the request from a system dependent operation into system independent terms. A simple dialog such as GetColor may be converted to a portable form merely by converting arguments to portable structures as shown below.

Mac
Boolean GetColor(Point Where,
*   String255 Prompt,*
*   RGBColor *Selection,*
*   RGBColor *Default);*
Portable
boolean GetTheColor(Point Where,
*   const char *Prompt,*
*// convert to portable char **
*   COLOR *Selection,*
*// convert to portable COLOR **
// (see above)
*   COLOR *Default);*
*// convert to portable COLOR **
// (see above)

File requests follow the same principle but are somewhat more complex. The portable version of a file name and a directory is a string indicating the full path. The extension may be used on the PC as a version of the file type and may be used to filter data. The PC and X Windows do not support creator type as a separate parameter. Under Windows, using a unique extension for each creator type allows icons to be assigned correctly. The two general file operations supported are retrieving the name of an existing file and selecting the name of a new file, called as follows.

```
boolean GetExistingFile(char **Name,
       char* StartDirectory,
       char **TheCreator);
```
where Name starts as the default name and returns as the selected full path name. Directory starts as the directory, and TheCreator returns as the Creator on the Mac and the Extension on the PC. GetExistingFile is implemented in the Mac as a call to SFGetFile using a routine to interconvert a frefnumber and a vrefnumber pairs into path names.

```
boolean NewFileName(char **Name,
       char* StartDirectory,
       char **TheCreator);
```

is a similar portable call to create a new file.

One feature requiring the modification of Macintosh dialogs is a call to return the name of a folder. This dialog provides that service on all systems. It converts to a modified SFGetFile dialog on the Mac and a similarly modified dialog on the PC.

*char *GetExistingDirectory (char *StartDirectory);*

These levels of abstraction are important to allow a library to use high level dialogs without tying the code to any one system.

Summary

Development of portable code poses unique and interesting challenges. Portable applications look very different from applications developed to run on only one platform. The application must be written from the ground up to maximize the fraction of the code that is system independent. The resulting application will involve compromises and will not be the best and most efficient application for any one system.

References

Steven M. Lewis , Cognits: A Portable Library of Intelligent Classes, Macintosh Developers Conference 1992.
Martin Minnow, Mouse Track and Field I, MacTutor 6:2,1990
Martin Minnow, Mouse Track and Field I, MacTutor 6:3,1990