Guided tour of a generic HID USB Windows Application

J.B.Gilmer Wilkes University Oct 30, 2010

This document walks the reader though a look at a relatively simple generic HID USB application that uses check boxes (for LED's and Switches) and Edit boxes (for text reporting accelerometer values and such).

Below is from the application .h file (in my case frictionD.h). What you see here is the definition of a C++ "class" for the application "FrictionD" itself. "CFrictionDApp" (the application class for my application) is a subclass of "CWinApp. The function by the same name, "`CFrictionDApp()`" is the "creator" function. It's public so Windows can start it up when you double click on its icon or otherwise start it up. It also has a function "DECLARE_MESSAGE_MAP()" that helps connect it to Windows.

```
class CFrictionDApp : public CWinApp
{
public:
	CFrictionDApp();

// Overrides
	public:
	virtual BOOL InitInstance();

// Implementation

	DECLARE_MESSAGE_MAP()
};

extern CFrictionDApp theApp;
```

Within the Aplication C file we find the code for the initialization function. It really doesn't do anything, the parent class CWinApp has the functions that actually set things up, so you don't have to do much. You do need to save the space for the application though, and write the code for "init_instance() which does the initialization for your particular kind of application.

At the top we have some critical includes. Stdafx is key: it hooks you into all the functions for windows, controls like buttons, edit boxes, and so forth. More about that later. "Dlg" stands for "Dialog. That's the box you see on the screen with which the user interacts. (You could have an App with no dialog. Usually you have onethough.)

```
#include "stdafx.h"
#include "FrictionD.h"
#include "FrictionDDlg.h"
```

Then the code to create the Application:

```
CFrictionDApp::CFrictionDApp()
{
	// TODO: add construction code here,
	// Place all significant initialization in InitInstance
}

// The one and only CFrictionDApp object
```

```
CFrictionDApp theApp;
```

Here's init_instance():

```
BOOL CFrictionDApp::InitInstance()
{
	// InitCommonControlsEx() is required on Windows XP if an application
	// manifest specifies use of ComCtl32.dll version 6 or later to enable
	// visual styles.  Otherwise, any window creation will fail.
	INITCOMMONCONTROLSEX InitCtrls;
	InitCtrls.dwSize = sizeof(InitCtrls);
	// Set this to include all the common control classes you want to use
	// in your application.
	InitCtrls.dwICC = ICC_WIN95_CLASSES;
	InitCommonControlsEx(&InitCtrls);
	CWinApp::InitInstance();
	AfxEnableControlContainer();
	// Standard initialization
	SetRegistryKey(_T("Local AppWizard-Generated Applications"));
	CFrictionDDlg dlg;
	m_pMainWnd = &dlg;
	INT_PTR nResponse = dlg.DoModal();
	// Any response in "Done" box should exit for now
	return FALSE;
}
```

A lot of this is "just Windows". It's important, you have to do it, but we don't want to get into the details. Notice that we call "Init_instance" for our parent class, CWinApp, so all the Windows stuff gets initialized too. We have to initialize "controls" (buttons like quit, resize, and all that stuff we inherit from CWinApp). Two very important lines are:

```
	CFrictionDDlg dlg;
	m_pMainWnd = &dlg;
```

This is there we declare the dialog (box) and refer to it by its class name of CFrictionDDlg, and allocate a variable by which we will know it: "dlg". Then, we put a pointer to the dialog at m_pmainWnd, a variable for the pointer to the "main window" for any CWinApp. Whenever we need a pointer to the dialog, that's where we will find it. And we'll need to do that often. I'm skipping some of the comments in the code that are stuff we don't need to consider.

Now, let's look into the header file where the dialog is defined, in my case FrictionDDlg.h.

```
class CFrictionDDlg : public CDialog
{
private:
	UINT_PTR timer;
// Construction
public:
	CFrictionDDlg(CWnd* pParent = NULL);	// standard constructor
// Dialog Data
	enum { IDD = IDD_FRICTIOND_DIALOG };

	protected:
	virtual void DoDataExchange(CDataExchange* pDX);// DDX/DDV support
```

We see that it "inherits" from the standard Windows "CDialog" class. The constructor function is "public" so that the App can create one of these things; that's really all the App does. The IDD and DoDataExchange are "Windows stuff" that facilitates getting the needed information into our dialog when the user does something. We'll see how those get used a bit later. Notice also that we save space for a timer. It's private; we'll only use it from within the dialog.

```
// Implementation
protected:
	HICON m_hIcon;

	// Generated message map functions
	virtual BOOL OnInitDialog();
	afx_msg void OnSysCommand(UINT nID,LPARAM lParam);
	afx_msg void OnPaint();
	afx_msg HCURSOR OnQueryDragIcon();
	DECLARE_MESSAGE_MAP()
```

The above handle the usual stuff Windows dialog boxes have to do; we won't go into details. For example, "OnPaint()" is the function that repaints the dialog whern it comes back to the foreground. Thankfully, we don't have to write code for that; it's all handled by the CDialog class that we inherit from. But, we can add code to this is we need to.

```
public:
	afx_msg void OnBnClickedCheck1();
	afx_msg void OnBnClickedDone();
	……(I'm skipping a bunch of similar stuff)
	CButton cbLed1;
	CButton cbLed2;
	……
	CButton cbSw2;
	……
	afx_msg void OnBnClickedCheck8();
	afx_msg void OnBnClickedSw3();
	……
	CEdit   ceText1;
	afx_msg void OnEnUpdateValue1();
	CString m_TextString;
	int SamplingActive;
	int FileActive;
	……
	CEdit   ceTextTime;
	……
	afx_msg void OnEnChangeValueTime();
	afx_msg void OnEnUpdateValueTime();
};
```

Here we have the stuff that we need to put into the header file for the control objects (in my case CButtons and CEdits) that will appear on the screen in the dialog box. So, CButton cbSw2; creates in my dialog a switch box object that can be checked or unchecked. I need to declare functions that I'm going to use to do something when a user does something. For a switch box or button, that's "OnBnClickedSw3()" for when the user clicks on switch3 (the CButton object named cbSw3). The associations between the code and the actions will come later. Notice that these are "afx_msg"

functions; you find them in afxwin.h. So, if you have a "control" and you want to know what you can do with it, look at that header file. (It's huge; several thousand lines.)

So, now that we see what's in our dialog object, we can now look at the details of those functions, as found in the .cpp file. (Mine is named FrictionDDlg.cpp, corresponding to the header file name.)

First, we have includes, the same ones as before, plus "hid_dev.h" so we can get to the critical usb communications functions. I also included stdio so I could have a couple of text files that I would write to, one for debugging purposes (outfile) and the other to write out data for the user's analysis. Later I'll take out most of the debug file writes.

```
#include "stdafx.h"
#include "FrictionD.h"
#include "FrictionDDlg.h"
#include "hid_dev.h"
#include <stdio.h>  //for debugging purposes
FILE *outfile;      //debug output
FILE *dataoutfile;  // Data output file
```

Next we have some globals. Two length variables will be reported back from hid_dev.c for our USB output and input messages, to tell us how many bytes of data we can send and receive. Then we have three functions. These are NOT part of the dialog; they are just freestanding functions we can call from within this file. I use one of these to update the high order led's, one for the low order ones, and the third for updating the switches and other information in the dialog when a message comes back from the microcontroller. Indeed, these are where most of the work gets done.

```
// Global variables from hid_dev.cpp
extern DWORD OutputReportByteLength;
extern DWORD InputReportByteLength;

static void update_sws (void);
static void update_leds_low(void);
static void update_leds_high(void);
```

Next we have a "callback". This is sort of like an interrupt service routine for the microcontroller. It gets initiated from outside our App when something happens at the user ort windows level. Notice the "UINT_PTR idEvent" that will tell us what kind of event caused the callback. When we get the callback, we will update the switches (and other stuff).

```
//This timer callback procedure will be called periodically.
//It will update sw1 and sw2 state.
VOID CALLBACK my_t_proc(HWND hwnd, UINT uMsg,  UINT_PTR idEvent, DWORD
dwTime)
{
  update_sws();
}
```

Now, finally, we hav the “creator” for the dialog itself. (The function name is the same as the class name; that’s how you know it’s a creator. You can create one by asking for a “new” object of that kind, or declare one to exist as an object (as we did in our App).

```
CFrictionDDlg::CFrictionDDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CFrictionDDlg::IDD, pParent)
{
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

We pretty much leave this alone; it just sets the icon or something; most of the stuff is done by CDialog and we don’t have to fool with it. Following this, we have the code for the DoDataExchange function. This is where we associate the name of the control (in the dialog resource) with the objet in our Dialog object. Don’t make a mistake here! You get very opaque Windows error exits.

```
void CFrictionDDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
  DDX_Control(pDX, IDC_CHECK1, cbLed1);
  DDX_Control(pDX, IDC_CHECK2, cbLed2);
  ……
  DDX_Control(pDX, IDC_VALUE10, ceTextTime);
}
```

Then we have the message map. This is where we connect the different objects with the functions we declared in our dialog to be called when the objects get clicked, edited, or whatever by the user. Don’t make a mistake on these; the error messages are hard to associate with where the error actually is.

```
BEGIN_MESSAGE_MAP(CFrictionDDlg, CDialog)
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	//}}AFX_MSG_MAP
	ON_BN_CLICKED(IDC_CHECK1, &CFrictionDDlg::OnBnClickedCheck1)
	ON_BN_CLICKED(IDC_CHECK2, &CFrictionDDlg::OnBnClickedCheck2)
	……
	ON_EN_CHANGE(IDC_VALUE10, &CFrictionDDlg::OnEnChangeValueTime)
	ON_EN_UPDATE(IDC_VALUE10, &CFrictionDDlg::OnEnUpdateValueTime)
END_MESSAGE_MAP()
```

Now we get to “our” code, the code that actually does the dialog stuff, starting with initialization when the dialog gets started. Right at the beginning we call the OnInitDialog for our parent class CDialog so it can get set up right, then we do our own stuff. That means first setting the values of important global variables. (Gee; I wonder where these were defined?) We want to start with one of the check boxes checked, so we check it. Within a dialog function, we can just refer to the member object and dot call it’s function. From one of the functions outside the dialog this is more complicated, as we will see.

```
// CFrictionDDlg message handlers
BOOL CFrictionDDlg::OnInitDialog()
{
```

```
    CDialog::OnInitDialog();
  //Initialize some variables
   FileActive=0;
   SamplingActive=0;
  // Indicate auto V control
   cbSw5.SetCheck(BST_CHECKED);
```

Next we open the files for debugging and data out mentioned earlier. The "MessageBoxW" function is a convenient way to warn the user what has gone wrong. Here we use it if we can't open the file for some reason. Notice that we immediately write a message to the file. After doing that, fflush makes sure what we wrote actually gets flushed into the file in case out program crashes before the buffer is emptied. This really helps if you are crashing somewhere; you'll get what was last written before the crash.

```
//Open Debug output file  jbg
    outfile=fopen("outfile.txt","w");
    if(outfile==0){
     MessageBoxW(L"outfile initialization failed.", L"Error", MB_OK);
     exit(1);
  }
    fprintf(outfile,"FrictionDDlg debug output file\n\n");
    fflush(outfile);
```

Then we have some necessary Windows stuff:

```
    // Set the icon for this dialog.  The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);            // Set big icon
    SetIcon(m_hIcon, FALSE);           // Set small icon
```

And now we open up the USB connection. If we can't find the connection, we use MessageBoxW to tell the user, and we quit.

```
    //Initialize HID module.
    if (HIDOpen())  // Was Stubbed out.  This is what I need hid_dev.h for!
    {
          MessageBoxW(L"HID initialization failed.", L"Error", MB_OK);
          exit(1);
    };
```

Now that the USB port is open, we send a message to turn off all of the LEDs. At the PC end, we don't have to wait until the earlier message has cleared to send a new one. In my case, I need two messages. One has a high order hex digit of "2" and the other "3" to tell the microcontroller what to do with the bottom 4 bits. (I can only send 8 bit messages, and there are lots of different things I want to be able to send.) After doing that, I set the timer to deliver the callback, and I'm done with initialization.

```
    //Turn off all LEDs (changed from 1 byte to 8 bytes)
    unsigned char c;
    c=0x20;
    HIDWrite(&c); //zero LED's 1-4
    c=0x30;
    HIDWrite(&c); //zero LED's 5-8
```

```
	timer=SetTimer(1, 10, my_t_proc);
	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Next, there is some stuff for dragging and paining that doesn't need to be changed.

```
void CFrictionDDlg::OnPaint()
{
……
}
HCURSOR CFrictionDDlg::OnQueryDragIcon()
{
	return static_cast<HCURSOR>(m_hIcon);
}
```

Now we get to the code that does stuff. Here's the code to update the low order LED's. The critical thing to notice is how we get the pointer to the dialog. We go back to the application object and get it's pointer to the dialog we set up when we created the window. Notice here dlg is a pointer, not the dialog itself. (I'd have used pdlg, but I'm building off the code in the sample.) Whenever this function gets called, we send out a message to write the LED's, just as we sent out a message to write them all to zero in InitInstance just earlier. In my case, I put "2" in the upper digit to indicate the low order LEDs are the four to be written, with the values I get from the check boxes in the dialog. Notice I'm writing to the debug file so I can see what's going on. But, how does this function get called? We'll see pretty soon. (There's also a similar function for the high order LEDs omitted in this document.)

```
//This function will update LEDS on the board to reflect the state
//of GUI controls.  This one does the low order LEDs.
static void update_leds_low(void)
{
  unsigned char lstate=0;
  CFrictionDDlg *dlg;
  dlg=(CFrictionDDlg *)theApp.m_pMainWnd;
  if (dlg->cbLed1.GetCheck())lstate |= 1;
  if (dlg->cbLed2.GetCheck())lstate |= 2;
  if (dlg->cbLed3.GetCheck())lstate |= 4;
  if (dlg->cbLed4.GetCheck())lstate |= 8;
  char outbuf;
  outbuf=lstate|0x20;
  HIDWrite(&outbuf);
  fprintf(outfile,"Write LED's 1-4 %x\n");
  fflush(outfile);
}
```

Now we have the code for the switch update function. You may recall that this gets called by our callback function (earlier). The first part updates the switch box indicators to correspond with data coming in the first byte of the "in" message. But, now, I have the in message at 8 bytes so I can also pass data for the accelerometer and display it in EditBoxes in the dialog.

```
//This function will update GUI control state to reflect the state
//of sw1 to sw4 on the board.
```

```
static void update_sws(void)
{
    unsigned char datain[9];
    unsigned char lstate=0;
    unsigned char c;
    static unsigned char highbyte;
    unsigned char reportcount;
  CFrictionDDlg *dlg;
  dlg=(CFrictionDDlg *)theApp.m_pMainWnd;
  if (HIDRead(datain))
  {
    // A message has come in; put allthe data where it needs to go.
    //switches are in low 4 bits of datain[0]
    lstate=datain[0]&0x0f;
    if (lstate & 0x1)
    {
      dlg->cbSw1.SetCheck(BST_CHECKED);
    }
    else
    {
      dlg->cbSw1.SetCheck(BST_UNCHECKED);
    }
    if (lstate & 0x2)
    {
```

…… (similar for the other switches)

```
      dlg->cbSw4.SetCheck(BST_UNCHECKED);
    }
```

If it was just the switches, we could return now. But, I want to update editboxes for the other data too. The functions of CEdit are found in afxwin.h (found in VS9\VC\altmfc\include). First, for each of the three edit boxes, I select all of the test in the box so I can replace it later. To select it all, I have to get the end for each. Notice that here also we have to use the pointer to the dialog to get at the respective control objects within it.

```
    // Reset text to zero length
    int nend=dlg->ceText1.LineLength(-1);
    fprintf(outfile,"LineLength returns %d\n",nend);
    dlg->ceText1.SetSel(0,nend,FALSE); //TRUE?
    nend=dlg->ceText2.LineLength(-1);
    dlg->ceText2.SetSel(0,nend,FALSE); //TRUE?
    nend=dlg->ceText3.LineLength(-1);
    dlg->ceText3.SetSel(0,nend,FALSE); //TRUE?
```

For debugging, I'm printing the incoming message so I can figure out what is going on.

```
    //Get input data
    int i, x;
    char string[20];
    fprintf(outfile," datain values:");
    for(i=0;i<9;i++){x=datain[i]; fprintf(outfile," %d",x);}
    fprintf(outfile,"\n");
```

Next, I convert the value of the X variable (two bytes) and print it.

```
    // Convert X into floating point and display it.
    x=datain[1]*256+datain[2];
    float xf;
    xf = x*5./4096.;
    fprintf(outfile,"x value calculated as %d  %6.3f\n",x, xf);
```

At this point we have the floating value x. We even have a string with the value of x in it. But, to print it to the screen, we need x as a "Unicode" string. There are functions that are supposed to convert, but I have not been able to get them to work. C is a nice language. Yoyu can do things by brute force even when computer science people would tell you you shouldn't do it that way. So, that's what I did. For Unicode, all you have to do is to convert each character into a word with an upper zero byte. You want other languages? Then it gets messier. But this works. Oh, and if "FileActive" we also put the x data into our output file for the user. FileActive is a declared member of the duialog apparently. Once the value of x is converted to Unicode, we use "ReplaceSel" to replace whatever had previously been in the Edit Box with the new value (in Unicode). Then we do the same for y, z, and a bunch of other stuff.

```
    //What we'd really like here is a Unicode sprintf.
    //Don't know how to convert otherwise, So, have to write our own code to
do this.
    sprintf(string,"%6.3f",xf);
    if(dlg->FileActive!=0)fprintf(dataoutfile,"%6.3f, ",xf);
    char wstring[42];
    //MultiByteToWideChar(0,0,string,-1,wstring,20); // couldn't get this to
work.
    for(i=0; i<20&&string[i]!=0;i++){
            wstring[i*2+1]=0;
            wstring[i*2]=string[i];}
    wstring[2*i]=0;
    wstring[2*i+1]=0;
    LPCTSTR wstr=(LPCTSTR) wstring;
    //fprintf(outfile,"wstring from x string:");
    //for(i=0;i<10;i++){x=wstring[i]; fprintf(outfile," %2x",x);}
    //fprintf(outfile,"\n");
    dlg->ceText1.ReplaceSel(wstr,FALSE);
    ……
```

So, next we have the member functions that respond to the "messages" coming from Windows for the various things the user can do. This one is for the user clicking on the box for an LED. All it does is call the function to update the LED's. All four low order click boxes do the same thing; they all four update the bottom four LEDs.

```
void CFrictionDDlg::OnBnClickedCheck1()
{
      // TODO: Add your control notification handler code here
       update_leds_low();
}
```

The functions for the switches don't do anything! That's because we don't expect the user to click them. He can, but it won't do no good. Our own program to update the switches when we get a message in takes care of these.

```
void CFrictionDDlg::OnBnClickedSw1()
{
	// TODO: Add your control notification handler code here
	//Do nothing more
}
```

One of the buttons is the “done” box to exit:

```
void CFrictionDDlg::OnBnClickedDone()
{
	exit(FALSE);
}
```

The Edit boxes for x, y, and z likewise are not for user input; they are output only. So, their handlers don’t need to do anything either. We write to them as part of update_switches().

```
void CFrictionDDlg::OnEnChangeValue1()
{
	// TODO:  If this is a RICHEDIT control, the control will not
	// send this notification unless you override the
CDialog::OnInitDialog()
	// function and call CRichEditCtrl().SetEventMask()
	// with the ENM_CHANGE flag ORed into the mask.
	// TODO:  Add your control notification handler code here
	//nothing needed; this will be read only
}
void CFrictionDDlg::OnEnUpdateValue1()
{
	// TODO:  If this is a RICHEDIT control, the control will not
	// send this notification unless you override the
CDialog::OnInitDialog()
	// function to send the EM_SETEVENTMASK message to the control
	// with the ENM_UPDATE flag ORed into the lParam mask.
	// TODO:  Add your control notification handler code here
}
```

One of my edit boxes is different; it’s for the user to type in a value that gets sent (as a series of hex digits) to the microcontroller. So, instead of writing to the Editbox, we need to see what’s there. Of course, it’s in Unicode and needs to be converted into ascii for normal C character handling.

```
void CFrictionDDlg::OnEnChangeValue4(){
  char outbuf;
  CFrictionDDlg *dlg;
  char wstring[42], str[20];
  LPTSTR lpszbuffer=(LPTSTR) wstring;
  int nend,x,i;
  dlg=(CFrictionDDlg *)theApp.m_pMainWnd;
  nend=dlg->ceText4.LineLength(-1);
  if(nend==0)x=0;
  else{
      dlg->ceText4.GetLine(0,lpszbuffer);
      fprintf(outfile,"send speed buffer:");
      for(i=0;i<40;i++){
         x=wstring[i];
```

```
         fprintf(outfile," %d",x);}
      fprintf(outfile,"\n");
      for(i=0;i<20;i++)str[i]=wstring[2*i];
      sscanf(str,"%i",&x);
  }
```

Once the value has been obtained, it's written to the debug file, and then it is passed to the microcontroller 4 bits at a time via 3 messages to convey all 12 bits.

```
  fprintf(outfile,"speed= %d",x);
  fflush(outfile);
  outbuf=(x&0x0f)|0x40;
  HIDWrite(&outbuf);
  outbuf=((x&0xf0)>>4)|0x50;
  HIDWrite(&outbuf);
  outbuf=((x&0xf00)>>8)|0x60;
  HIDWrite(&outbuf);
}
```

The key to using Dialogs, Edit Boxes, Check boxes and all the other stuff is to know what functions you can call for them. Sometimes it's hard to find the functions because they are somewhere in a hierarchy of classes between "Window" and the particular kind of object. For example, if you find the class:

```
class CWnd : public CCmdTarget
```

This one goes on for many pages, all commands that pertain to objects thaty are the target of a command. Another important type of CWin is the dialog. This one has lots of functions and variables too. Most of this is Windows stuff you don't have to fool with (ansd I'm skipping some.)

```
/////////////////////////////////////////////////////////////////////////////
// CDialog - a modal or modeless dialog
class CDialog : public CWnd
{
      DECLARE_DYNAMIC(CDialog)
      // Modeless construct
public:
      CDialog();
      virtual BOOL Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL);
      virtual BOOL Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);
      virtual BOOL CreateIndirect(LPCDLGTEMPLATE lpDialogTemplate, CWnd*
                              pParentWnd = NULL,
      void* lpDialogInit = NULL);
      virtual BOOL CreateIndirect(HGLOBAL hDialogTemplate,
                              CWnd* pParentWnd = NULL);
……
// Attributes
public:
      void MapDialogRect(LPRECT lpRect) const;
      void SetHelpID(UINT nIDR);
// Operations
public:
```

```
	// modal processing
	virtual INT_PTR DoModal();
	……
	// default button access
	void SetDefID(UINT nID);
	DWORD GetDefID() const;
	// termination
	void EndDialog(int nResult);
	// Overridables (special message map entries)
	virtual BOOL OnInitDialog();
	virtual void OnSetFont(CFont* pFont);
protected:
	virtual void OnOK();
	virtual void OnCancel();
// Implementation
public:
	virtual ~CDialog();
	……
	virtual BOOL PreTranslateMessage(MSG* pMsg);
	virtual BOOL OnCmdMsg(UINT nID, int nCode, void* pExtra,
		AFX_CMDHANDLERINFO* pHandlerInfo);
	virtual BOOL CheckAutoCenter();
protected:
	UINT m_nIDHelp;		// Help ID (0 for none, see HID_BASE_RESOURCE)
	// parameters for 'DoModal'
	LPCTSTR m_lpszTemplateName;	// name or MAKEINTRESOURCE
	……
protected:
	//{{AFX_MSG(CDialog)
	afx_msg LRESULT OnCommandHelp(WPARAM wParam, LPARAM lParam);
	afx_msg LRESULT OnHelpHitTest(WPARAM wParam, LPARAM lParam);
```

Here near the end we have some of the functions we need to use for our code:

```
	afx_msg LRESULT HandleInitDialog(WPARAM, LPARAM);
	afx_msg LRESULT HandleSetFont(WPARAM, LPARAM);
	afx_msg void OnPaint();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

Likewise, we can find the functions for buttons and edit boxes. Some are subclasses of CStatic (which we won't detail):

```
class CStatic : public CWnd
```

Here's the one we are most directly using: CButton

```
class CButton : public CWnd
{
	DECLARE_DYNAMIC(CButton)
// Constructors
public:
	CButton();
	virtual BOOL Create(LPCTSTR lpszCaption, DWORD dwStyle,
```

```
                        const RECT& rect, CWnd* pParentWnd, UINT nID);
// Attributes
      UINT GetState() const;
      void SetState(BOOL bHighlight);
      int GetCheck() const;
      void SetCheck(int nCheck);
      UINT GetButtonStyle() const;
      void SetButtonStyle(UINT nStyle, BOOL bRedraw = TRUE);
      HICON SetIcon(HICON hIcon);
      HICON GetIcon() const;
      HBITMAP SetBitmap(HBITMAP hBitmap);
      HBITMAP GetBitmap() const;
      HCURSOR SetCursor(HCURSOR hCursor);
      HCURSOR GetCursor();
#if (_WIN32_WINNT >= 0x501)
      AFX_ANSI_DEPRECATED BOOL GetIdealSize(_Out_ LPSIZE psize) const;
      AFX_ANSI_DEPRECATED BOOL SetImageList(_In_ PBUTTON_IMAGELIST
pbuttonImagelist);
      AFX_ANSI_DEPRECATED BOOL GetImageList(_In_ PBUTTON_IMAGELIST
pbuttonImagelist) const;
      AFX_ANSI_DEPRECATED BOOL SetTextMargin(_In_ LPRECT pmargin);
      AFX_ANSI_DEPRECATED BOOL GetTextMargin(_Out_ LPRECT pmargin) const;
#endif  // (_WIN32_WINNT >= 0x501)
#if ( _WIN32_WINNT >= 0x0600 ) && defined(UNICODE)
      CString GetNote() const;
      _Check_return_ BOOL GetNote(_Out_z_cap_(*pcchNote) LPTSTR lpszNote,
_Inout_ UINT* pcchNote) const;
      BOOL SetNote(_In_z_ LPCTSTR lpszNote);
      UINT GetNoteLength() const;
      BOOL GetSplitInfo(_Out_ PBUTTON_SPLITINFO pInfo) const;
      BOOL SetSplitInfo(_In_ PBUTTON_SPLITINFO pInfo);
      UINT GetSplitStyle() const;
      BOOL SetSplitStyle(_In_ UINT nStyle);
      BOOL GetSplitSize(_Out_ LPSIZE pSize) const;
      BOOL SetSplitSize(_In_ LPSIZE pSize);
      CImageList* GetSplitImageList() const;
      BOOL SetSplitImageList(_In_ CImageList* pSplitImageList);
      TCHAR GetSplitGlyph() const;
      BOOL SetSplitGlyph(_In_ TCHAR chGlyph);
      BOOL SetDropDownState(_In_ BOOL fDropDown);
      // Sets whether the action associated with the button requires elevated
permissions.
      // If elevated permissions are required then the button should display
an elevated icon.
      HICON SetShield(_In_ BOOL fElevationRequired);
#endif // ( _WIN32_WINNT >= 0x600 ) && defined(UNICODE)
// Overridables (for owner draw only)
      virtual void DrawItem(LPDRAWITEMSTRUCT lpDrawItemStruct);
// Implementation
public:
      virtual ~CButton();
protected:
      virtual BOOL OnChildNotify(UINT, WPARAM, LPARAM, LRESULT*);
};
```

Notice how many things you can do with a button! Some things that might be useful include highlighting the button, GetCheck(), and SetCheck(). In our code we actually use only

the last two. Many of these are for things that are compiled into the resource when we set up our dioalog box. But, we could use these functions to dynamically create a button on the fly if we wanted to, or even make one go away.

The other class we need to use is CEdit. This is a very flexible class we could use for a large box of text if we wanted to. A lot of this at the beginning is essential Windows stuff.

```
class CEdit : public CWnd
{
	// DECLARE_DYNAMIC virtual OK - CWnd already has DECLARE_DYNAMIC
	DECLARE_DYNAMIC(CEdit)
// Constructors
public:
	CEdit();
	BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT
nID);
// Attributes
	BOOL CanUndo() const;
	int GetLineCount() const;
	BOOL GetModify() const;
	void SetModify(BOOL bModified = TRUE);
	void GetRect(LPRECT lpRect) const;
	DWORD GetSel() const;
	void GetSel(int& nStartChar, int& nEndChar) const;
	HLOCAL GetHandle() const;
	void SetHandle(HLOCAL hBuffer);
	void SetMargins(UINT nLeft, UINT nRight);
	DWORD GetMargins() const;
	void SetLimitText(UINT nMax);
	UINT GetLimitText() const;
	CPoint PosFromChar(UINT nChar) const;
	int CharFromPos(CPoint pt) const;
	// NOTE: first word in lpszBuffer must contain the size of the buffer!
	// NOTE: may not return null character
	int GetLine(_In_ int nIndex, _Out_ LPTSTR lpszBuffer) const;
	// NOTE: may not return null character
	int GetLine(_In_ int nIndex, _Out_cap_post_count_(nMaxLength, return)
LPTSTR lpszBuffer, _In_ int nMaxLength) const;
#if (_WIN32_WINNT >= 0x501)
	AFX_ANSI_DEPRECATED BOOL SetCueBanner(_In_z_ LPCWSTR lpszText, _In_
BOOL fDrawIfFocused = FALSE);
	AFX_ANSI_DEPRECATED BOOL GetCueBanner(_Out_z_cap_(cchText) LPWSTR
lpszText, _In_ int cchText) const;
#if defined(UNICODE)
	CString GetCueBanner() const;
	BOOL ShowBalloonTip(_In_z_ LPCWSTR lpszTitle, _In_z_ LPCWSTR lpszText,
_In_ INT ttiIcon = TTI_NONE);
	BOOL ShowBalloonTip(_In_ PEDITBALLOONTIP pEditBalloonTip);
	BOOL HideBalloonTip();
#endif  // (UNICODE)
#endif  // (_WIN32_WINNT >= 0x501)
#if (_WIN32_WINNT >= 0x0600) && defined(UNICODE)
	// REVIEW: Sets the characters in the edit control that are
highlighted.
	void SetHighlight(_In_ int ichStart, _In_ int ichEnd);
```

```
	// REVIEW: Retrieves the characters in the edit control that are
highlighted.
	BOOL GetHighlight(_Out_ int* pichStart, _Out_ int* pichEnd) const;
#endif  // (_WIN32_WINNT >= 0x0600) && defined(UNICODE)
```

From here we see a long list of operations we can perform on an edit box. We can even support undo and redo, and pasting stuff to and from the clipboard! (I never have bothered.) You can also create or resize boxes on the fly. You will see here the operations I used to write stuff to and read stuff from my small Edit boxes.

```
// Operations
	void EmptyUndoBuffer();
	BOOL FmtLines(BOOL bAddEOL);
	void LimitText(int nChars = 0);
	int LineFromChar(int nIndex = -1) const;
	int LineIndex(int nLine = -1) const;
	int LineLength(int nLine = -1) const;
	void LineScroll(int nLines, int nChars = 0);
	void ReplaceSel(LPCTSTR lpszNewText, BOOL bCanUndo = FALSE);
	void SetPasswordChar(TCHAR ch);
	void SetRect(LPCRECT lpRect);
	void SetRectNP(LPCRECT lpRect);
	void SetSel(DWORD dwSelection, BOOL bNoScroll = FALSE);
	void SetSel(int nStartChar, int nEndChar, BOOL bNoScroll = FALSE);
	BOOL SetTabStops(int nTabStops, LPINT rgTabStops);
	void SetTabStops();
	BOOL SetTabStops(const int& cxEachStop);    // takes an 'int'
	// Clipboard operations
	BOOL Undo();
	void Clear();
	void Copy();
	void Cut();
	void Paste();
	BOOL SetReadOnly(BOOL bReadOnly = TRUE);
	int GetFirstVisibleLine() const;
	TCHAR GetPasswordChar() const;
// Implementation
public:
	// virtual OK here - ~CWnd already virtual
	virtual ~CEdit();
};
```

You may decide to use other controls. This is where you go to find out what you can do with them. Ultimately, there are lots of things you can do. The Horton text has a good bit of stuff about graphics that you can do in a smaller window within your dialog box, and you can even create additional windows, documents, and much else. But, this is as far as I expect us to go inthis course.

I hope this “guided tour” of a simple “generic USB” application helps.