Using Edit boxes in a Windows C++ Application

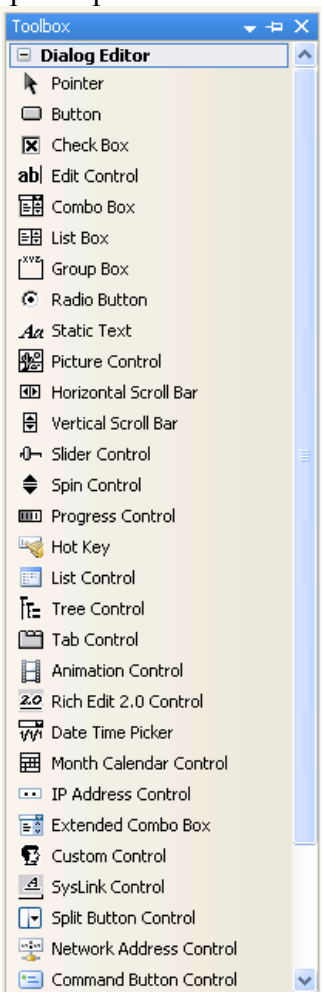
October 18, 2012

Purpose

From the sample Windows application for interacting with the microcontroller by USB as a Human Interface device one can find examples of click boxes and buttons (really both the same sort of thing) that can serve as examples. But to get and put text onto the screen, or read text from the screen, is more involved. This document is intended to be an aid in understanding how to do that. It will illustrate the process step by step. It is assumed that the reader is already familiar somewhat with using Visual Studio 2008 and with the Freescale / CMX USBLITE hid example code.

Resource editor

The first step in adding a text cap-ability is to add the text object to the application's dialog. The dialog is represented in the project as a resource and as the code to go with that resource. The first step is to add an edit box to the resource. There are several kinds of things you could add. Open up the resource for the dialog box. You will find a tab for "Toolbox" that can be opened to list all the sorts of things that can be pasted into the dialog. We will use an "IEdBoxEditor" Look on the Toolbox list and double click on "Edit Control". Then in the dialog sketch out the rectangle you want the edit box to occupy. Once it's inplace, right click and bring up "Properties".



In the Properties block you can set all sorts of things, but for most you can use the default. One thing you need to do is assign or record the ID. It will be of the form "IDC_VALUE12" or something like that. (See the figure on the previous page.) You can change this to be more meaningful, but you need to remember it so that you can connect this resource to the code for it in your dialog. This is where you can also add freestanding text labels, images, and other embellishments to your dialog box that don't have a programmed function. You are now finished with adding the resource, so save and close the dialog box resource.

Dialog header file

The next step is to edit your dialog header file to add the corresponding edit box object to the dialog object. The dialog header file (title should be "AppNameDlg.h") defines a class "AppNameDlg "for your dialog, as a subclass of a Windows "CDialog" class. You need to add an "Edit Control".

Here's the question: How do you know what the objects are that you can insert here, and what functions they use to interact with your code? The includes for the dialog code file include "stdafx.h", the standard include file for Windows applications. (If you search for instances of this file, you'll find a lot of them. But, there should be one actually in your project – note that the name is in quotes, not brackets, so it's local.) This header file links you to all the Windows stuff, including <afxcmn.h> : Windows common controls, and <afxwin.h> Core and standard components. These files and others at C:\Program Files\Microsoft Visual Studio 9.0\VC\atlmfc\include. In afxwin.h you will find definitions of all various controls and other objectsand the functions you can use with them. One of these is CEdit. There is a "Rich Edit Control" also that has more features that is in afxcmn.h. But we'll use the simpler CEdit.

What we need to know is the class of the text object. It is a "CEdit." So, under "public:" in the Dialog class (in the dialog header file) we need to add:

```
CEdit   ceTextVoltage;
```

This adds an object of that type to our dialog. We can use any name we want. I generally use ceText as the first 6 letters to indicate it is a "CEdit" and that I'm using it for text. The rest of the name indicates what I'm using it for.

The next issue is, do I need functions in the Dialog for the user interacting with this text box? For example, if the user is going to type stuff in, I need a function, called a "method" or "message" in C++ lingo, that will handle the event that occurs. This would be a callback from Windows to my method / function when the user types something in. Let's provide for that. We will add a function to the Dialog for it:

```
afx_msg void OnEnChangeValueVoltage();
```

Putting this in obligates me to write this function and put it in the dialog C++ file later. An afx_msg is a callback, and it returns nothing (void) back to Windows and passes in nothing. (This is like the terminal callbacks for different commands in the terminal demo program for the microcontroller, but it hooks to a control in Windows instead of a text terminal string.)

While you are here, note the other things that can be found in the dialog class.

Dialog C++ code file

Open up the dialog C++ file, AppNameDlg.cpp. Inside this file are all the functions defined in the dialog header file (and some other miscellaneous stuff too). Notice that the callback for the timer is what updates the switches (indicating which ones are pressed and which ones are not).

The first dialog function is the "creator". Notice that it has the form "AppNameDlg::ApNameDlg(CWnd* pParent):CDialog(AppNameDlg::IDD, pParent){…}. This is a typical C++ trick based on "inheritance"; the application dialog is a subclass of the Windows CDialog (generic dialog), so the "creator" function (that has the name of the class) is just calling the creator for the generic class object. What is passed in is a pointer to the Window – a dialog is really just a subclass of Window. Nothing here has to change. Notice that the "::" notation is used with the class for which the function is a member on the left, and the function name is on the right. Different classes (that is, types of objects) can have functions with the same name and that's not a problem, since each is applicable only to that class.

There is a dialog function for "Do Data Exchange". This has the form:

AppNameDlg::DoDataExchange(CDataExchange* pDX){….}

A "CDataExchange" is actually one of the objects embedded in the dialog. (Perhaps a member of the parent CDialog class?) It manages interactions between the Windows representation (from the resource) and your C++ code (the corresponding objects). So, in this function a function call is needed to get the data back and forth. Add a function to do this for the new CEdit:

DXX_Control(pDX, IDC_VALUE12, ceTextVoltage);

Notice that this is where you link the name of the resource to the variable, that is, the CEdit object within your application's dialog object. Be sure you don't make a typo here; it may not be detected.

Next, for any of the function of your new CEdit, you need to link that also to the Windows resource. The MESSAGE MAP does that. Find "BEGIN MESSAGE MAP(….." and add a function (for the parent CDialog object) that indicates which of your functions to call. Here we would put:

ON_EN_CHANGE(IDC_VALUE12, &AppNameDlg:: OnEnChangeValueVoltage)

Notice that this message map stuff isn't a normal C++ function. It's a "Macro" like what a "#DEFINE" does, but can get very complicated and arbitrary. The all capital letters is a tipoff. We are not going to try to figure out how it works. Notice that the punctuation isn't the same as for a C function. Note there that what we are doing is connecting a pointer to our function (added to the dialog.h file) up to the condition detected in the dialog for the new edit box.

Now, what we need to add is the function itself. We need a new function:

AppNameDlg:: OnEnChangeValueVoltage(){ ……. }

This is the function that responds to the user making a change in the new "Voltage" edit box. If you just leave it empty, nothing happens. The user will be able to type text into the box. It is there to be read is somewhere your code wants to see what is written. But the function is called whenever a change occurs. So, if you only want to do something on such occasions, use this function. If you want to 'poll" the value regularly, change or not, then you don't need this function (and would not add it to the dialog or add this method and would not need to put it in the message map). Whether you need such a function, this is how it would be done.

Dialog code to write and read text

Now that you have an edit box, you will want to have your program write stuff to it. There are two different contexts in which this could happen:

1) A member function of the dialog –in a function like AppNameDlg::OnBnClickedCheck4()
2) A function that is not a member function, like update_leds() in the example code.

The difference is that inside a member function you automatically have access to all member variables of the Dialog. You don't need a pointer to the Dialog itself; that is assumed. Things are a bit simpler. Inside an arbitrary function no such access is predefined. You have to provide for everything. The example code for the led's and buttons takes the latter approach, which is helpful in illustrating what needs to happen. In adding access to the edit box we will assume the same for now.

So, suppose we want to write the string "5 Volts" into the box. Here's what we have to do:

Declarations needed:

```
AppNameDlg *dlg;  //This defines a local variable that will be a pointer to our dialog.
int nend;                 //a variable for how long the text in the dialog is
char wstring[42]={'5',0,' ',0,'V',0,'o',0,'l',0,'t',0,'s',0,0,0,0,0}; //"5 Volts" in Unicode
LPCTSTR wstr=(LPCTSTR) wstring;
```

Executable statements:

```
dlg = (AppNameDlg *)theApp.m_pMainWnd;  //Gets the pointer to our dialog, puts it in dlg.
nend = dlg->ceTextVoltage.LineLength(-1);       //Finds the line length of the text in the box
dlg->ceTextVoltage.SetSel(0,nend,FALSE);     //Selects all the current text in the box
dlg-> ceTextVoltage.ReplaceSel(wstr,FALSE); //Replaces the selection with new text
```

This takes some explanation. We need a local variable for a pointer to the dialog because we are not in a dialog member function, so the first declaration and the first executable statement take care of that. Inside a member function you'd omit these, and skip the "dlg->" in all the function calls.

The problem with string handling is that Windows requires "Unicode" 16 bit characters. An array of these is a "LPCTSTR". So, what we need to do is create a string with ascii characters in the even character positions and 0's in the odd character positions, then tell Windows that this is an LPCSTR. So, that's what we do. The variable "wstring" is an ordinary ascii character string. It points to an array of 42 8 bit character variables. We then create the variable "wstr" which is a pointer to Unicode 16 bit characters, and sety it to point to the same array of characters. When we access these characters with our code we will use "wstring". When we tell Windows functions about it, we use "wstr". In this case, the string is pre-loaded.

The form "LPCTSTR wstr=(LPCTSTR) wstring;" is a "cast" – we are taking a pointer to one kind of thing and saying it is a pointer to a different kind of thing. C lets you do that. Some other "stricter" languages don't, Pascal for example. If you are in languages like that and can't find the approved solution to how to do something, you are truly stuck. C lets you find a work – around, as I have here. You'd really like a function that will convert ascii strings to Unicode strings. I never found one I was able to get to work, so I used this expedient, which works.

So, the remaining executable statements correspond almost exactly to what you would do with a mouse on the screen. First you look to see how much text is already in the window. (That is, you find the length of the string and put it in "nend". Then you select (and highlight) all the text, from 0 to nend. Then you replace what's selected with new text.

How do you know the names of the functions you can use and what they do? That's all in "afxwin.h". CEdit is a subclass of CWnd (a C Window) which is a subclass of CCmdTarget (An object that can be the target of commands). There are LOTS of functions you can use with a CEdit, including things like setting margins, setting up help balloons, highlighting, and much more. Also, the CEdit text box supports the clipboard editing functions: cut, copy, paste etc. The other function you are most likely to need is GetSel(), to read what's there.

Suppose you wanted to write to the edit box the value of a variable. That is tricky since you have to convert the value to Unicode. Here's how you can do it:

```
#include <stdio.h>  // required to use "sprintf" function
……
char string[20];     //Where the string will come from
int i;
float x;             //the variable to be written
int nend;              //a variable for how long the text in the dialog is
char wstring[42];  //The char string for where we will put the Unicode string
LPCTSTR wstr=(LPCTSTR) wstring;  //The pointer to the Unicode string
…….
sprintf(string,"x = %6.3f\0", x);  //prints the value of x into the string
for(i=0;i<20&&string[i]!=0;i++){  //copy the string
        wstring[i*2+1]=0;            //zeros into odd characters
        wstring[i*2]=string[i];}     //string characters into even characters
wstring[2*i]=0;                      //Pads it out with extra zeros
wstring[2*i+1]=0;
nend = dlg->ceTextVoltage.LineLength(-1);   //Finds the line length of the text in the box
dlg->ceTextVoltage.SetSel(0,nend,FALSE);     //Selects all the current text in the box
dlg-> ceTextVoltage.ReplaceSel(wstr,FALSE); //Replaces the selection with new text
```

Notice that the last three lines are the same as before. The difference is that we start with a variable rather than a constant Unicode string. Getting the value from a CEdit is the reverse of this:

```
dlg->ceTextVoltage.GetLinel(0,wstr);     //Gets the text in the box
for(i=0;i<40&&wstring[i]!=0;i++){
        string[i]=wstring[2*i];}
string[i]=0;
sscanf(string,"%f",&x);  //we are assuming here that we want the value as a float
```

Debugging (Be sure to read this)

Visual Studio 2008 includes a debugger. But, one of the most reliable (if primitive) methods of debugging is to simple use print statements. On the microcontroller our ability to do that is limited by the fact that we don't have a file system. If we are using the terminal we can do a "print" but that's of limited use if we are using the hid interface because the terminal's slow speed and spin-wait methods of delay timing interfere with proper usb operation. (I've tried it!) But on the PC under Windows we do have an operating system and files, so we can take advantage of that. All we have to do is #include <stdio.h> and we can write stuff out.

Here's my recommendation: inside the dialog creator, before trying to do HIDOpen(), crerate and open a file to dump debug information into. Somewhere near the top of AppNameDlg.cpp put:

```
#include <stdio.h>
…….
FILE *outfile;
```

Then, in the dialog initializer before trying to do HIDOpen, you can put:

```
outfile = fopen("outfile.txt","w");
fprintf(outfile,"AppName debug output /n");
fflush(outfile);
```

This opens a file named "outfile.txt" for write "w" only. Whenever you want to write something to the file, you can use the fprintf function. Stuff you write goes into a print buffer, and it only gets dumped out into the file occasionally, so it's possible that if your program crashes you won't get all the stuff that's been written. The function fflush() takes care of that, and writes everything out so that you will have it all.

Important: If you look in the distributed (modified) copy of hid_dev.cpp that needs to be linked into your project, you will find that I've inserted "debug writes". I may have commented these out, but possibly not. If they are still there, either comment them out or include the debug file into which stuff can be written. I needed this when I was debugging the hid interface the first time and needed to understand what was happening. This is the file where HIDOpen, HIDWrite, and HIDRead can be found, which are the functions used to open up communications with the DEMOJM, and send data out or get it back via USB. Notice at the top in HIDOpen() the constants for vid and pid. These need to match what the DEMOJM reports back to the USB driver when it is first connected, or this application on the PC won't find the device that it is looking for, and initialization will fail.

Other stuff:

There are all kinds of controls: sliders, "radio buttons", spin controls for example. You are welcome to try to include some of these. Sliders or spinners might be interesting for setting train speed (train motor voltage duty cycle) for example. You'd follow many of the same procedural steps described here, but the specifics of the messages and functions used to interact with the control would be different. My suggestion is to keep it simple until you have something working, and always keep earlier versions that you can go back to if things don't work out.