phos: A Console GUI Component

Michael Mangelsdorf
18 Aug 2009 CPOL 7 min read

A DLL implemented in assembler featuring a console GUI component

This project is the development of a DLL which encapsulates a visually pleasing, customizable text console component written in assembler. On startup, the DLL puts up its application window which the client of the DLL can control using a simple API including a printstr statement, without any knowledge about the Win32 API.

You have an auto-saved draft version. Would you like to view or edit it?

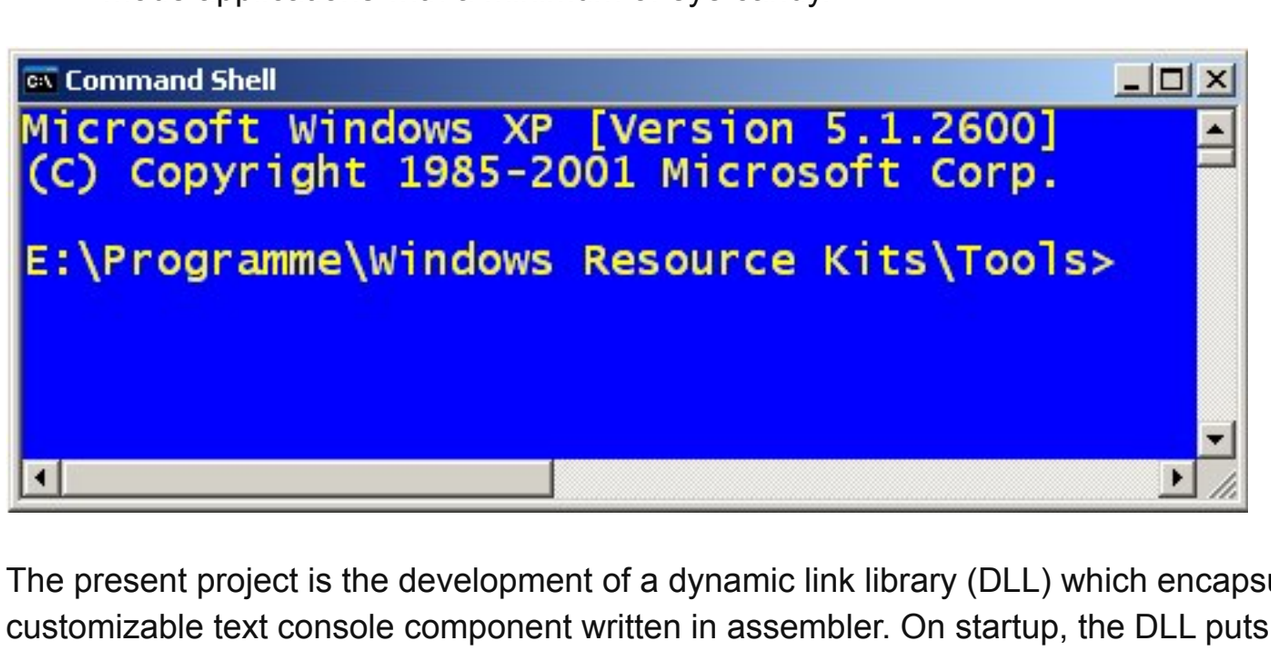
- Download demo project - 22 KB
- Download source - 35.5 KB

Introduction

When developing non-managed Windows applications in C, assembler, or other languages using the Win32 API, linkers usually produce either a *console* or a *Windows* type executable. Console applications automatically display a Windows console window similar to *CMD.EXE*, whereas true Windows applications are fully responsible for constructing their own GUI using Win32 API calls.

Both options put unnecessary constraints on the casual programmer, which this contribution is intended to compromise.

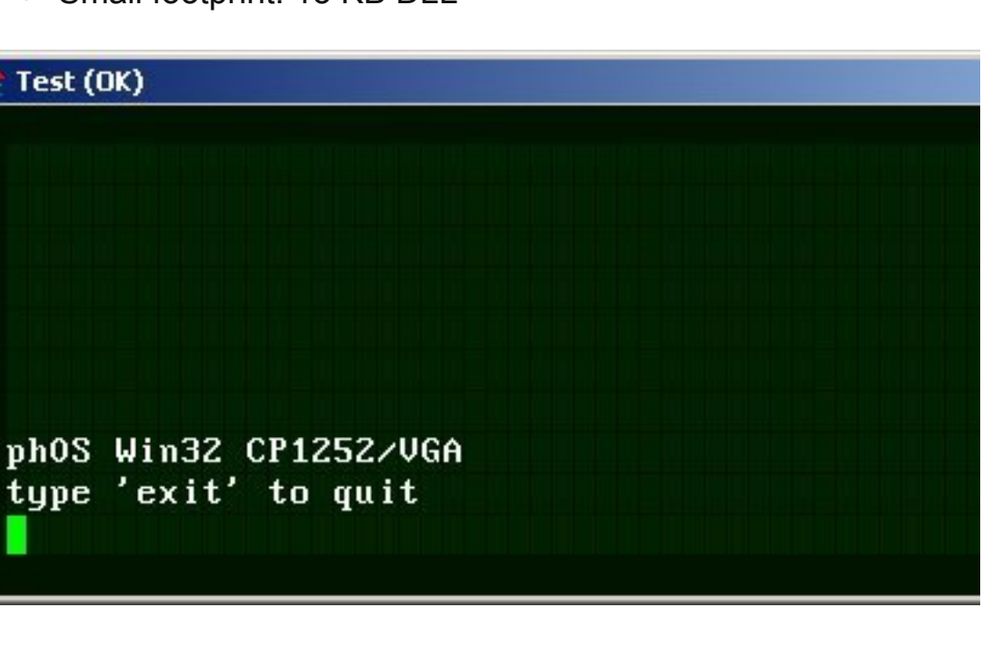
- Many programmers will agree that console applications have a substandard, generic look and feel to them (see the blue screenshot) which is not always acceptable for programmers or users.
- Windows application development requires a range of programmer skills (event driven programming, use of API calls, GDI, etc.) which casual programmers might not expect to be necessary for contributing text mode applications with a minimum of eye candy.



The present project is the development of a dynamic link library (DLL) which encapsulates a visually pleasing, customizable text console component written in assembler. On startup, the DLL puts up its application window (see dark screenshot) which the client of the DLL can control using a simple API including a `printstr` statement, without any knowledge about the Win32 API.

Main features of the GUI window include:

- Resizable borders to adjust how many lines are displayed
- Customizable color scheme
- Provide your own icon
- Callback functions for initialization, prompt line available, and window close events
- Window stays fully responsive while DLL client processes prompt line (runs in own thread)
- Small footprint: 13 KB DLL



Required Background

The first part of the article will describe the C demo program. To compile the demo source, you need LCC-Win32, free for non-commercial use (http://www.cs.virginia.edu/~lcc-win32/).

The source is self explanatory, and should not present difficulties even if you have never used a DLL before.

The second part of the article is about the internal workings of the DLL. If you wish to understand that part, you should have a basic understanding of the Win32 API and the i386 assembler. To build the DLL from source, you need MASM32 (http://www.masm32.com/).

Part 1 - DLL Client (C Demo)

How do you interface with the DLL?

The intrinsics of attaching to the DLL are hidden in the functions included in the *phos.h* header file. All you need as a user of the DLL is to keep the DLL "visible" to your *.EXE* file. The basic steps, however, to make functions contained in the DLL available to your program are as follows:

C++ Copy Code
```
// define a pointer variable to a function
void (APIENTRY *pfn_phos_vscroll)(void);

// load the DLL
dllHandle = LoadLibrary( "phos32.dll" );

// fill the pointer variable with the effective address
pfn_phos_vscroll = GetProcAddress( dllHandle, "phos_vscroll" );

//call the function using its pointer
pfn_phos_vscroll();
```

Walk-through

Let's quickly go through the whole sample application, excluding the header file. You can see that the main function is very short. Firstly, a custom icon gets loaded. The DLL functions are made available in a single function call, and then the actual application window is produced.

The arguments to `pfn_phos_start_window()` are pointers to the other three functions contained in *test.c*. They are the so-called callback functions.

1. `phos_thread_func()` is called whenever the user of the window hits Return, submitting a line of input. If your program takes longer than 500 ms to process the input, the title bar of the window shows a running count of the seconds elapsed.
2. `phos_init_callback()` is only called once, namely when the window is initially shown. If you are familiar with the Win32 API, this function is called when the message loop of the window is already set up.
3. `phos_exit_callback()` is also only called once, when the window is closing for some reason (the user has clicked the cross, or hit **ALT-F4**, etc). This function is for cleaning up tasks.

C++ Copy Code
```
#include <stdio.h>
#include <stdlib.h>

#include "phos.h"

///////////////////////////////////////////////////////////////////////////////

int main(int argc, char *argv[])
{
  HICON hIco;

    hInstMain = GetModuleHandle(NULL);

    // if you don't provide an icon, pass NULL
    // to use phos default icon from DLL

    hIco = LoadIcon( hInstMain, MAKEINTRESOURCE(APP_ICON) );

    if (load_DLL_functions()) {
        pfn_phos_start_window( phos_thread_func,
                               phos_init_callback,
                               phos_exit_callback,
                               hIco );
    }
    return 0;
}
```

The next section of code defines the two callback functions that are only run once. You can see that we use `phos_init_callback()` to set a suitable window caption, the color scheme, and sample output.

C++ Copy Code
```
///////////////////////////////////////////////////////////////////////////////

// phos_init_callback() is called once, when the phos console
// window is displayed (WM_CREATE handler)

void phos_init_callback ( HANDLE hWin, HANDLE hInstDll )
{
    pfn_phos_set_caption( "Test" );

    pfn_phos_set_colorscheme ( PETROL_GREEN );

    pfn_phos_printstr("phOS Win32 CP1252/VGA", 0xFFFFFF, 0);
    pfn_phos_vscroll();

    pfn_phos_printstr("type 'exit' to quit", 0xFFFFFF, 0);
    pfn_phos_vscroll();
}

///////////////////////////////////////////////////////////////////////////////

// phos_exit_callback() is called once, when the phos console
// window is about to close (WM_DESTROY handler)

void phos_exit_callback()
{
}
```

The remaining part of *test.c* defines `phos_thread_func()`, the heart of the demo program. This function is called whenever a line of text is submitted at the console prompt. The argument we are getting is a C structure with three important values, defined in *phos.h*.

`thread_info->ptrLineBuffer` is simply a pointer to a zero terminated string with the line entered. We store it in `cmd` to abbreviate the notation of it.

`thread_info->reason` is the Windows virtual keycode that was entered to submit the input line. This will most often be `VK_RETURN`, the **Enter** key, but can also be `VK_UP` or `VK_DOWN` (up and down arrow keys), in case you wish to implement some BASH style command line history.

Finally, let's look at the following interesting statement in detail:

C++ Copy Code
```
pos = pfn_phos_printstr("Thread says hello! ", 0xFFFFFF, 0);
```

The hexadecimal value is the requested RGB value for the text color. The final zero means that the string will be printed beginning at column zero. The variable `pos` gets the position of the next character *after* the output line after the `string` has been printed. This is to be able to chain calls to this function and build up a line successively.

C++ Shrink ▲ Copy Code
```
///////////////////////////////////////////////////////////////////////////////

// phos_thread_func() is the parameter to CreateThread
// when phos.dll creates the user thread

void phos_thread_func( struct phos_thread_info* thread_info)
{
  char* cmd = thread_info->ptrLineBuffer;
  int pos;

    if (thread_info->reason == VK_RETURN)
    {
        if (strlen(cmd) != 0)
        {
            if (strcmp("exit", cmd) == 0)
            {
                Sleep(500);
                // setting lparam of the message to 1 -> quit application
                PostMessage( thread_info->hWin, WM_FINISH, 0, 1);
            }
            else
            {
                pfn_phos_vscroll();
                pos = pfn_phos_printstr("Thread says hello! ", 0xFFFFFF, 0);
                pos = pfn_phos_printstr( cmd, 0x00FFFF, pos);
            }
        }
    }
    pfn_phos_vscroll();
    PostMessage( thread_info->hWin, WM_FINISH, 0, 0 );
}
```

Don't delete the final line - it sends an important message to the application window, telling it that your program has finished processing the input line.

Part 2 - DLL Source (MASM32)

If you are just interested in *using* the DLL, you may skip down to the end of the article at this point, as the following section is concerned with how the DLL does its job.

While the assembler source is reasonably well documented, let's look at the following points in some detail:

- Line buffer vs. bitmap
- GDI, font bitmap, backbuffer bitmap, device context bitmap

Line Buffer vs. Bitmap

Developing a low level GUI application that displays text exposes a distinction between the logical text in a buffer (`a + b + c = abc`) and the physical text (bitmaps) as a matrix of pixels.

Phos deals with this distinction in the following way.

Font Bitmap, Backbuffer Bitmap, and Device Context Bitmap

The DLL uses three different bitmaps to generate a window output in a two step process:

1. The required character cells are copied from the character bitmap to the appropriate places of the backbuffer bitmap. The character bitmap is a viewable bitmap image 16 pixels high and 255 x 8 = 2040 pixels wide, containing an 8x16 pixel character cell for each character.
2. Only when the window receives a `WM_PAINT` message, indicating that the window needs (re)painting, the visible portion of the backbuffer is copied onto the screen bitmap in a call to `BitBlt`. Because this is an integral operation, flicker is much reduced as opposed to the naive approach of just copying the characters directly on the screen bitmap. This is a common technique.

A variable called `lineBuffer` contains the current input line as a zero terminated string. One group of functions dealing with cursor motion, insertions, etc., exclusively deals with this variable.

Another group of functions affects the `backbuffer` bitmap by copying characters from the character bitmap onto it, or by copying blocks of pixels between different parts of it.

The following figure shows the two step process of copying bitmaps and the bitmaps involved.

DLL Operation

The DLL uses the following basic operating steps:

- Provides a DLL skeleton
- Sets up the GDI context, initializes bitmaps
- Starts a message loop
- Listens for Windows messages, particularly `WM_SIZING` and `WM_CHAR`
- Starts a registered user function in a separate thread once the Return key is pressed
- Deregisters resources and closes the window

As a general example, the following shows the code for the `WM_SIZING` Windows message handler in `WndProc`.

ASM Shrink ▲ Copy Code
```
.elseif uMsg == WM_SIZING                       ;user is pulling window border(s)

    ;without this message handler, the user can just resize the window
    ;however she likes - this becomes ugly quickly because during WM_PAINT,
    ;we only paint the visible portion of our back buffer bitmap;
    ;if the window is extended beyond that, we only see the desktop
    ;background

    ;the message sends us the newly requested coordinates of the window
    ;called the sizing rectangle;
    ;we can prevent the resizing by simply overwriting the coordinates in the
    ;sizing rectangle by what was there before

    ;we are essentially preventing the user from resizing horizontally, and
    ;from making the window taller than the maximum number of lines, but also
    ;showing less than the minimum number of lines

    ;once the sizing is over (mouse released), the WM_EXITSIZEMOVE handler
    ;is called, snapping the size of the window to the nearest line of text

    ;note that what you see here is dependent on the Windows option:
    ;"Show window contents while dragging"
    ;(right click on desktop, choose properties, appearance, effects)

    ;*****************************
    ;handle horizontal sizing;
    ;prevent resizing by simply
    ;replacing requested size
    ;by current size
    ;*****************************

    mov eax, lParam

    mov ecx, (RECT PTR [eax]).right
    sub ecx, (RECT PTR [eax]).left              ;ecx current sizing rect width

    mov edx, OFFSET SizingRect

    mov ebx, (RECT PTR [edx]).right
    sub ebx, (RECT PTR [edx]).left              ;ebx previous sizing rect width

    .if wParam == WMSZ_RIGHT       || \         ;pulling right border or corners?
        wParam == WMSZ_BOTTOMRIGHT || \
        wParam == WMSZ_TOPRIGHT

        mov ecx, (RECT PTR [eax]).left          ;current left plus
        add ecx, ebx                            ;current width = current right
        mov (RECT PTR [eax]).right, ecx         ;overwrite requested right

    .elseif wParam == WMSZ_LEFT       || \      ;pulling left border or corners?
            wParam == WMSZ_BOTTOMLEFT || \
            wParam == WMSZ_TOPLEFT

        mov ecx, (RECT PTR [eax]).right         ;current right plus
        sub ecx, ebx                            ;current width = current left
        mov (RECT PTR [eax]).left, ecx          ;overwrite requested left

    .endif

    ;*****************************
    ;handle vertical sizing
    ;prevent resizing by
    ;replacing requested size
    ;by current size
    ;*****************************

    mov ecx, (RECT PTR [eax]).bottom            ;eax lParam
    sub ecx, (RECT PTR [eax]).top               ;ecx requested window height

    mov ebx, (RECT PTR [edx]).bottom
    sub ebx, (RECT PTR [edx]).top               ;ebx current window height
    cmp ecx, ebx                                ;check whether requested height
    jg @F                                       ;less (shrink) or
                                                ;greater (grow) than current
                                                ;height

    ;CASE 1   ;***************
              ;shrinking window
              ;***************

    mov edx, paddedClientHeightCurrent
    add edx, ecx
    sub edx, ebx                                ;result: new/projected height
    cmp edx, paddedClientHeightMin              ;would the sizing request
    jle adjust_shrinking                        ;get us into the minimum zone?

    mov paddedClientHeightCurrent, edx          ;nothing to do - just update

    mov ecx, (RECT PTR [eax]).top
    mov ebx, (RECT PTR [eax]).bottom

    jmp end_sizing

adjust_shrinking:                               ;we are inside the minimum zone;
                                                ;to avoid overshooting, we do
                                                ;as if the user wanted exactly
                                                ;the minimum height, not less

    sub edx, paddedClientHeightMin              ;number of pixels the request is
                                                ;inside the minimum

    .if wParam == WMSZ_TOP      || \            ;top border downwards?
        wParam == WMSZ_TOPLEFT  || \
        wParam == WMSZ_TOPRIGHT

        mov ebx, (RECT PTR [eax]).top           ;adjust the top so that we are
        add ebx, edx                            ;just outside the minimum region
        mov (RECT PTR [eax]).top, ebx

    .elseif wParam == WMSZ_BOTTOM      || \     ;bottom border upwards?
            wParam == WMSZ_BOTTOMLEFT  || \
            wParam == WMSZ_BOTTOMRIGHT

        mov ebx, (RECT PTR [eax]).bottom        ;adjust the bottom so that we are
        sub ebx, edx                            ;just outside the minimum region
        mov (RECT PTR [eax]).bottom, ebx

    .endif

    mov edx, paddedClientHeightMin              ;finally, adjust client height
    mov paddedClientHeightCurrent, edx

    mov ecx, (RECT PTR [eax]).top
    mov ebx, (RECT PTR [eax]).bottom

    jmp end_sizing

    ;CASE 2   ;*************
              ;growing window
              ;*************
@@:
    mov edx, paddedClientHeightCurrent
    add edx, ecx
    sub edx, ebx
    cmp edx, paddedClientHeightMax
    jge adjust_growing

    mov paddedClientHeightCurrent, edx          ;update

    mov ecx, (RECT PTR [eax]).top
    mov ebx, (RECT PTR [eax]).bottom

    jmp end_sizing

adjust_growing:                                 ;we are outside vertical range;
                                                ;to avoid overshooting, we do
                                                ;as if the user wanted exactly
                                                ;the maximum height, not more

    sub edx, paddedClientHeightMax              ;number of pixels the request is
                                                ;outside the maximum

    .if wParam == WMSZ_TOP      || \            ;top edge pulled up
        wParam == WMSZ_TOPLEFT  || \
        wParam == WMSZ_TOPRIGHT

        mov ebx, (RECT PTR [eax]).top           ;adjust the top so that we are
        add ebx, edx                            ;just inside the vertical range
        mov (RECT PTR [eax]).top, ebx

    .elseif wParam == WMSZ_BOTTOM      || \     ;bottom edge pulled down
            wParam == WMSZ_BOTTOMLEFT  || \
            wParam == WMSZ_BOTTOMRIGHT

        mov ebx, (RECT PTR [eax]).bottom        ;adjust bottom so that we are
        sub ebx, edx                            ;just inside the vertical range
        mov (RECT PTR [eax]).bottom, ebx

    .endif

    mov edx, paddedClientHeightMax              ;finally, adjust client height
    mov paddedClientHeightCurrent, edx

    mov ecx, (RECT PTR [eax]).top
    mov ebx, (RECT PTR [eax]).bottom

end_sizing:

    mov edx, OFFSET SizingRect
    mov (RECT PTR [edx]).top, ecx
    mov (RECT PTR [edx]).bottom, ebx

    invoke updateWindowCaption

    mov eax, 1
    ret
```

Conclusion

Feel free to comment with suggestions, bug reports, requests, etc. Time permitting, the author will try to respond / change the article or expand on particular features of the assembler source that are of interest to you. This is a freeware/public domain contribution - use for any purpose, but there is absolutely no warranty.

References

- Iczelion's Win32 Assembly Tutorials
- MASM32 forum thread introducing the project

History

- 17th August, 2009: Initial version

License

Share

About the Author

Michael Mangelsdorf
Software Developer
Germany
Watch this Member

I currently work as a self-employed language translator. Back in school, I learned what must have been VAX Basic - phosphor green code on a glass tube. After transitioning from a VAX terminal to a Sinclair ZX81, I eventually got an Apple IIe and taught myself 6502 machine language. The past 30-odd years since have seen projects in various languages, for example Paver, a project built aro...

show more

Comments and Discussions

Yep	TheRaven	19-Sep-11 2:44
Nice work	programmersmind	10-Sep-09 10:10
Nice work.	Tom Delany	25-Aug-09 23:31
Suggestion	dpisarciuc	18-Aug-09 14:25
Re: Suggestion	Michael Mangelsdorf	18-Aug-09 14:41