Custom Controls in Win32 API: Scrolling

Nov 3, 2015 13 min read words

C Windows Win32 Win64 Visual-Studio Dev Visual-Studio custom-controls VS2013

by Martin Mitáš
Contributor

90k Views 4.98/ 5

- Simple scrolling example - 174.2 KB
- Advanced scrolling example - 186.2 KB

Articles in this series

- Custom Controls in Win32 API: The Basics
- Custom Controls in Win32 API: The Painting
- Custom Controls in Win32 API: Visual Styles
- Custom Controls in Win32 API: Standard Messages
- Custom Controls in Win32 API: Control Customization
- Custom Controls in Win32 API: Encapsulation of Customized Controls
- Custom Controls in Win32 API: Scrolling

Introduction

Windows API directly supports scrollbars within every single window and in the todays article we are going to show how to take advantage of it.

Note that in `COMCTL32.DLL`, Windows also offers a standalone scrollbar control but we won't cover it here. Once you understand the implicit scrollbar support we will talk about, usage of the standlone scrollbar control becomes very simple and straightforward.

Non-Client Area

Before we start talking about the scrollbars, we need to know about the concept of non-client area. In Windows, every window (`HWND`) distinguishes its client and non-client area. The client area (usually) covers most (or all) of the window on the screen, and actually more or less all fundamental contents of controls is painted in it.

Non-client area is an optional margin around the client area which can be used for some auxiliary content. For top-level windows, this involves the window border, the window caption with the window title and buttons for minimizing, maximizing and closing the window, the menubar and a border around the window.

Also child windows can and quite often take use of the non-client area. In most cases, a simple border and possibly (if needed) the scrollbars are painted in it. Usually (i.e. unless overridden) if the control has a style `WS_BORDER` or extended style `WS_EX_CLIENTEDGE`, the control gets the border.

Similarly, if the control decides it needs a scrollbar, Windows reserves more space in the non-client area on right and/or bottom side of the control for the scrollbars.

This behavior for the border and scrollbars is implemented in the function `DefWindowProc()` which handles many messages:

- `WM_NCCALCSIZE` determines dimensions of the non-client area. The default implementation looks for example at the style `WS_BORDER`, extended style `WS_EX_CLIENTEDGE` and state of the scrollbars to do so.
- `WM_NCxxxx` counterparts of various mouse messages together with `WM_NCHITTEST` handle interactivity of the non-client area. In the case of child control this typically involves reaction on the scrollbar buttons and `DefWindowProc()` does this for us.
- `WM_NCPAINT` is called to paint the non-client area. Again, handler of the message in `DefWindowProc()` knows how to paint the border and the scrollbars.

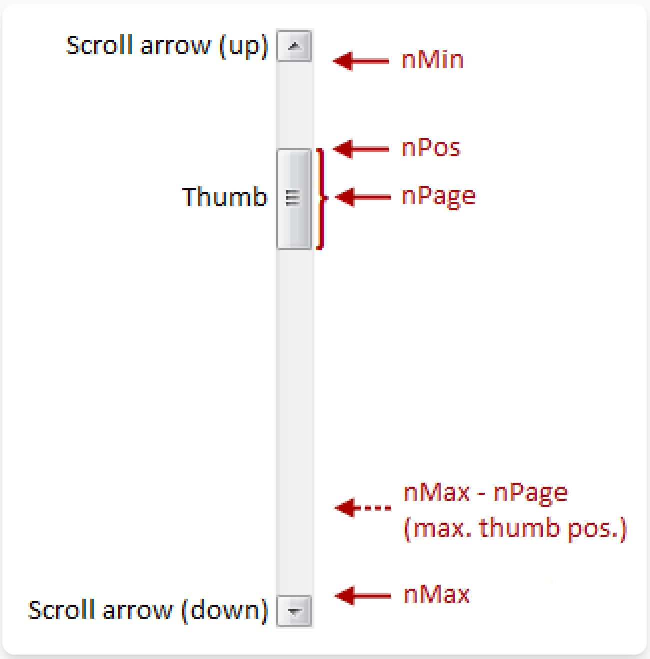
All of this standard behavior can be overridden if you handle these messages in your

Each HWND remembers two sets of few integer values which describe state of both the horizontal and vertical scrollbars. The set corresponds to the members of structure SCROLLINFO (except the auxiliary cbSize and fMask):

C

```c
typedef struct tagSCROLLINFO
{
    UINT    cbSize;
    UINT    fMask;
    int     nMin;
    int     nMax;
    UINT    nPage;
    int     nPos;
    int     nTrackPos;
}   SCROLLINFO, FAR *LPSCROLLINFO;
```

Note you are free to choose any units for the scrolling you like. Use whatever suits logic of your control the best. The values may be pixels, count of rows (or columns), amount of lines of text or whatever.



Anatomy of the scrollbar

The values nMin and nMax determine range of the scrollbars, i.e. minimal and maximal positions corresponding to the scrollbar's thumb moved to top (or left) and bottom (or right) position. In most cases nMin can be just always set to zero and control updates just the upper limit nMax.

The value nPage describes portion of the contents between nMin and nMax which can be displayed in the control given the size of its client area. Windows also visualizes this value in proportional size of the scrollbars thumb.

position. This value is read-only and cannot be directly changed programmatically.

Controls which want to support the scrolling can update these values with function `SetScrollInfo()`, or alternatively with some less general function like `SetScrollPos()` or `SetScrollRange()` which can update only subsets of the values.

Remember that all these setter functions implicitly ensure that `nPos` and `nPage` are always in the allowed ranges so that the following conditions hold all the time:

- `0 <= nPage < nMax - nMin`
- `nMin <= nPos <= nMax - nPage`

If it is logically impossible to fulfill the conditions, e.g. because `nPage > nMax - nMin`, then no scrolling is needed, Windows resets `nPos` to zero and hides the scrollbar (which results to the resizing of the client area and `WM_SIZE` message).

This means that you, as a caller of those function, do not need to care too much about the boundary cases. If, for example, you handle reaction to the key [PAGE UP] as scrolling a page up, you simply may do something like this:

C

```c
    // Get current nPos and nPage:
    int scrollbarId = (isVertical ? SB_VERT : SB_HORZ);
    SCROLLINFO si;
    si.cbSize = sizeof(SCROLLINFO);
    si.fMask = SIF_POS | SIF_PAGE;
    GetScrollInfo(hwnd, scrollbarId, &si);

    // Set new position one page up.
    // Note we do not care whether we underflow below nMin as SetScrollInfo()
    // does that for us automatically.
    si.fMask = SIF_POS;
    si.nPos -= si.nPage;
    SetScrollInfo(hwnd, scrollbarId, &si);

    // If we need to count with nPos below, we may need to ask again for the
fixed
    // up value of it:
    GetScrollInfo(hwnd, scrollbarId, &si);
```

new (visible) items are added or removed, or when an item is expanded or collapsed.

- Control has to update `nPage` when size of the client area changes (i.e. when handling `WM_SIZE`) so that it reflects amount of content which can fit in it.
- Control has to update `nPos` when it responds to the scrolling event as described by `WM_VSCROLL` or `WM_HSCROLL`. We will cover this more thoroughly later in this article.

Some other situations when the scrollbar state needs to be updated can be when dimension of some elements of the contents changes, e.g. when control starts to use different font which has different size. Often, the amount of related work depends how smartly you choose the scrolling unit: Consider a tree-view control and vertical scrolling: If it uses pixels as the scrolling units, then change of item height (e.g. as a result of `WM_SETFONT`) has to be reflected by recomputing of the scrollbar's state, but if you use rows as the scrolling units, then it does not.

Little Gotcha

When your control supports both horizontal and vertical scrollbars, there is a little trap. Remember that when setting up e.g. a vertical scrollbar, and the values change so that the scrollbar gets visible or gets hidden, the size of its client area changes.

This change in client area size can result also in the need to update state of the other scrollbar.

Consider the following code demonstrating the issue:

C

```c
static void
CustomOnWmSize(HWND hWnd, UINT uWidth, UINT uHeight)
{
    SCROLLINFO si;

    si.cbSize = sizeof(SCROLLINFO);
    si.fMask = SIF_PAGE;

    si.nPage = uWidth;
    SetScrollInfo(hWnd, SB_HORZ, &si, FALSE);

    // BUG: The SetScrollInfo() above can result in yet another resizing of
```

```c
    //
```

```
        // But after the recursive call returns the next call to SetScrollInfo()
        // may break the vertical scrollbar with possibly not-longer-valid value
        // of uHeight.

        si.nPage = uHeight;
        SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
    }

    static LRESULT
    CustomProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
    {
        switch(uMsg) {
            ...

            case WM_SIZE:
                CustomOnWmSize(hWnd, LOWORD(lParam), HIWORD(lParam));
                return 0;

            ...
        }
    }
```

Once you understand the issue, the fix is simple:

C

```
    static void
    CustomOnWmSize(HWND hWnd, UINT uWidth, UINT uHeight)
    {
        SCROLLINFO si;

        si.cbSize = sizeof(SCROLLINFO);
        si.fMask = SIF_PAGE;

        si.nPage = uWidth;
        SetScrollInfo(hWnd, SB_HORZ, &si, FALSE);

        // FIX: Make sure uHeight has the right value:
        {
```

```
        si.nPage = uHeight;
        SetScrollInfo(hWnd, SB_VERT, &si, TRUE);
    }
```

Handling WM_VSCROLL and WM_HSCROLL

When the scrollbar is visible (i.e. whenever `nMax - nMin > nPage`), and user interacts with it e.g. by clicking on a scrolling arrow button or by dragging the thumb, the window procedure gets corresponding non-client mouse messages. When passed to `DefWindowProc()`, they are translated to messages `WM_VSCROLL` (for the vertical scrollbar) and `WM_HSCROLL` (for the horizontal scrollbar).

The control's window procedure is supposed to handle them as follows:

1. Analyze the action requested by the user.
2. Update nPos accordingly.
3. Refresh client area so that the control presents corresponding portion of the contents.

Hence the typical handler code may look as follows:

C

```c
    static void
    CustomHandleVScroll(HWND hwnd, int iAction)
    {
        int nPos;
        int nOldPos;
        SCROLLINFO si;

        // Get current scrollbar state:
        si.cbSize = sizeof(SCROLLINFO);
        si.fMask = SIF_RANGE | SIF_PAGE | SIF_POS | SIF_TRACKPOS;
        GetScrollInfo(pData->hwnd, SB_VERT, &si);

        nOldPos = si.nPos;

        // Compute new nPos.
    case SB_LINEUP:         nPos = si.nPos - 1; break;
    case SB_LINEDOWN:       nPos = si.nPos + 1; break;
    case SB_PAGEUP:         nPos = si.nPos - CustomLogicalPage(si.nPage);
break;
    case SB_PAGEDOWN:       nPos = si.nPos + CustomLogicalPage(si.nPage);
break;
    case SB_THUMBTRACK:     nPos = si.nTrackPos; break;
    default:
    case SB_THUMBPOSITION:  nPos = si.nPos; break;
    }

    // Update the scrollbar state (nPos) and repaint it. The function
ensures
    // the nPos does not fall out of the allowed range between nMin and nMax
    // hence we ask for the corrected nPos again.
    SetScrollPos(hwnd, SB_VERT, nPos, TRUE);
    nPos = GetScrollPos(hwnd, SB_VERT);

    // Refresh the control (repaint it to reflect the new nPos). Note we
    // here multiply with some unspecified scrolling unit which specifies
    // amount of pixels corresponding to the 1 scrolling unit.
    // We will discuss ScrollWindowEx() more later in the article.
    ScrollWindowEx(hwnd, 0, (nOldPos - nPos) * scrollUnit
                   NULL, NULL, NULL, NULL, SW_ERASE | SW_INVALIDATE);
}

static LRESULT CALLBACK
CustomProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch(uMsg) {
        ...

        case WM_VSCROLL:
            // LOWORD(wParam) determines the desired scrolling action.
            CustomHandleVScroll(hwnd, LOWORD(wParam));
            return 0;

        ...
    }

    return DefWindowProc(hwnd, uMsg, wParam, lParam);
```

Updating the Client Area

In the code snippet above, we have used the function `ScrollWindowEx()`. Lets now take a closer look on it.

Painting the client area is task the control usually performs in the handler of message `WM_PAINT`. In case of control which supports scrolling, the function has to take the current `nPos` value into consideration. (Or actually two values, `nPosHoriz` and `nPosVert` if the control supports scrolling in both directions.)

Typically this means that the control contents is painted with vertical and horizontal offsets, `-(nPosVert * uScrollUnitHeight)` and `-(nPosHoriz * uScrollUnitWidth)`, so that the control presents content further to the bottom and right when the scrollbars are not in the minimal positions. (`uScrollUnitWidth` and `uScrollUnitHeight` determine width and height of the scrolling units in pixels.)

When application changes state of the scrollbar (i.e. the range, the position, or even the page size), it usually needs to repaint itself. It could just invalidate its client area and let `WM_PAINT` paint everything from scratch.

Or it can do something much smarter. In most cases when scrolling, there is often quite a lot of correctly painted stuff already available on the screen. It's just painted on bad position which corresponds to the old value of `nPos`, right?

The solution is to simply move all the still valid contents from the old position to the new one, and only invalidate portion of the client area which really needs to be repainted from scratch, i.e. only the area which roughly corresponds to the horizontal or vertical stripe which moves into the visible view-port from "behind the corner" during the scrolling operation.

And that is exactly what the function `ScrollWindowEx()` is good for. You tell it a rectangle, you tell it a horizontal and vertical offsets (difference between old and new `nPos`) in pixels, and it does all the magic. It actually copies/moves some graphic memory from one place to another to reuse as much as possible of the old contents, and it only invalidates those portions of the rectangle which really need to be repainted. Assuming the handler of `WM_PAINT` is implemented as it should and repaints only the dirty rectangle (refer to our 2nd part of this series about painting), it will then have much less work to do.

Scrolling with Keyboard

C

```c
static void
CustomHandleKeyDown(CustomData* pData, UINT vkCode)
{
    switch (vkCode) {
    case VK_HOME:   CustomHandleVScroll(pData, SB_TOP); break;
    case VK_END:    CustomHandleVScroll(pData, SB_BOTTOM); break;
    case VK_UP:     CustomHandleVScroll(pData, SB_LINEUP); break;
    case VK_DOWN:   CustomHandleVScroll(pData, SB_LINEDOWN); break;
    case VK_PRIOR:  CustomHandleVScroll(pData, SB_PAGEUP); break;
    case VK_NEXT:   CustomHandleVScroll(pData, SB_PAGEDOWN); break;
    }
}

static LRESULT CALLBACK
CustomProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch(uMsg) {
        ...

        case WM_KEYDOWN:
            CustomHandleKeyDown(pData, wParam);
            return 0;


        ...
    }

    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}
```

Scrolling with Mouse Wheel

Adding support for scrolling with a mouse wheel is somewhat more interesting. The main reason why it is not that simple is diversity of available hardware. The mouse wheel in many cases is not really a wheel, and often there is no mouse at all. Consider for example modern trackpads which may map some finger gestures to a virtual mouse wheel.

sometimes result in a larger delta coming at once, or a sequence of smaller deltas coming in short succession.

On Windows, the delta propagates as a parameter of the message `WM_MOUSEWHEEL` for vertical wheel, or `WM_MOUSEHWHEEL` for horizontal one: It is stored as the high word of `WPARAM`.

So, to handle these messages, application (or control in our case) has to accumulate the delta until it reaches some threshold value meaning "scroll one line down" (or up; or to left or right for horizontal scrolling).

Furthermore, the control should respect sensitivity of the wheel as configured in the system. On Windows, this settings can be retrieved with `SystemParametersInfo(SPI_GETWHEELSCROLLLINES)` for vertical wheel and `SystemParametersInfo(SPI_GETWHEELSCROLLCHARS)` for horizontal one. Both values correspond to the amount of vertical or horizontal scrolling units the control should scroll when the accumulated delta value reaches the value defined with macro `WHEEL_DELTA` (120).

The above may look quite difficult, but it's not that bad. Furthermore we may actually implement it just once: Windows supports only one mouse pointer and that implies there is never more then one vertical wheel and one horizontal wheel. Therefore we can use global variables for the accumulated values and one wrapping function dealing with them instead of bloating per-control data structures and reimplementing it in each window procedure.

The code of such function may look as follows:

C

```c
    // Lock protecting the static variables. Note you have to initialize the
    // critical section before calling the function WheelScrollLines() below
    // for the first time.
    static CRITICAL_SECTION csWheelLock;


    // Helper function for calculation of scrolling lines for provided mouse
wheel
    // delta value. This function is quite generic and can be used/shared among
    // many controls.
    // of SPI_GETWHEELSCROLLLINES and for some mouses.
    static HWND hwndCurrent = NULL;         // HWND we accumulate the delta
for.
    static int iAccumulator[2] = { 0, 0 };  // The accumulated value (vert.
and horiz.).
    static DWORD dwLastActivity[2] = { 0, 0 };

    UINT uSysParam;
    UINT uLinesPerWHEELDELTA;   // Scrolling speed (how much to scroll per
WHEEL_DELTA).
    int iLines;                 // How much to scroll for currently
accumulated value.
    int iDirIndex = (isVertical ? 0 : 1);  // The index into iAccumulator[].
    DWORD dwNow;

    dwNow = GetTickCount();

    // Even when nPage is below one line, we still want to scroll at least a
little.
    if (nPage < 1)
        nPage = 1;

    // Ask the system for scrolling speed.
    uSysParam = (isVertical ? SPI_GETWHEELSCROLLLINES :
SPI_GETWHEELSCROLLCHARS);
    if (!SystemParametersInfo(uSysParam, 0, &uLinesPerWHEELDELTA, 0))
        uLinesPerWHEELDELTA = 3;  // default when SystemParametersInfo()
fails.
    if (uLinesPerWHEELDELTA == WHEEL_PAGESCROLL) {
        // System tells to scroll over whole pages.
        uLinesPerWHEELDELTA = nPage;
    }
    if (uLinesPerWHEELDELTA > nPage) {
        // Slow down if page is too small. We don't want to scroll over
multiple
        // pages at once.
        uLinesPerWHEELDELTA = nPage;
    }

    EnterCriticalSection(&csWheelLock);
```

```
        iAccumulator[0] = 0;
        iAccumulator[1] = 0;
    } else if (dwNow - dwLastActivity[iDirIndex] > GetDoubleClickTime() * 2)
{
        // Reset the accumulator if there was a long time of wheel
inactivity.
        iAccumulator[iDirIndex] = 0;
    } else if ((iAccumulator[iDirIndex] > 0) == (iDelta < 0)) {
        // Reset the accumulator if scrolling direction has been reversed.
        iAccumulator[iDirIndex] = 0;
    }

    if (uLinesPerWHEELDELTA > 0) {
        // Accumulate the delta.
        iAccumulator[iDirIndex] += iDelta;

        // Compute the lines to scroll.
        iLines = (iAccumulator[iDirIndex] * (int)uLinesPerWHEELDELTA) /
WHEEL_DELTA;

        // Decrease the accumulator for the consumed amount.
        // (Corresponds to the remainder of the integer divide above.)
        iAccumulator[iDirIndex] -= (iLines * WHEEL_DELTA) /
(int)uLinesPerWHEELDELTA;
    } else {
        // uLinesPerWHEELDELTA == 0, i.e. likely configured to no scrolling
        // with mouse wheel.
        iLines = 0;
        iAccumulator[iDirIndex] = 0;
    }

    dwLastActivity[iDirIndex] = dwNow;
    LeaveCriticalSection(&csWheelLock);

    // Note that for vertical wheel, Windows provides the delta with
opposite
    // sign. Hence the minus.
    return (isVertical ? -iLines : iLines);
}
```

Notes:

scrolling unit up or down (SB_LINEUP and SB_LINEDOWN), or left or right (SB_LINELEFT and SB_LINERIGHT). Sorry for the terminology mess, but "scrolling units" is simply too much typing for someone as lazy as me...

- If the function is used in an application where different HWNDs are living in multiple threads, it has to be thread-safe to protect the state described by the multiple static variables. Hence the use of CRITICAL_SECTION.
- We reset the accumulators in certain situations: When HWND changes, when some longer time expires without the wheel activity or when user starts scrolling to the opposite direction. You may notice the period of inactivity is compared to a time period based on GetDoubleClickTime(). I chose to use that because the double-click time is used in Windows as a measure how good your reflexes are.
- For historic reasons, the delta value for the vertical wheel is provided with opposite sign then most people expect, and in the opposite sense in comparison to the horizontal wheel. To simplify the code we deal with that on the single spot: the last line of the function.

The function WheelScrollLines() is quite generic and reusable. Actually one could even think such function should be implemented in some standard Win32API library. That would at least provide better guaranty that mouse wheels are used consistently by default among applications. But AFAIK it is not, at least not a publicly exported one.

Usage of the function is very straightforward:

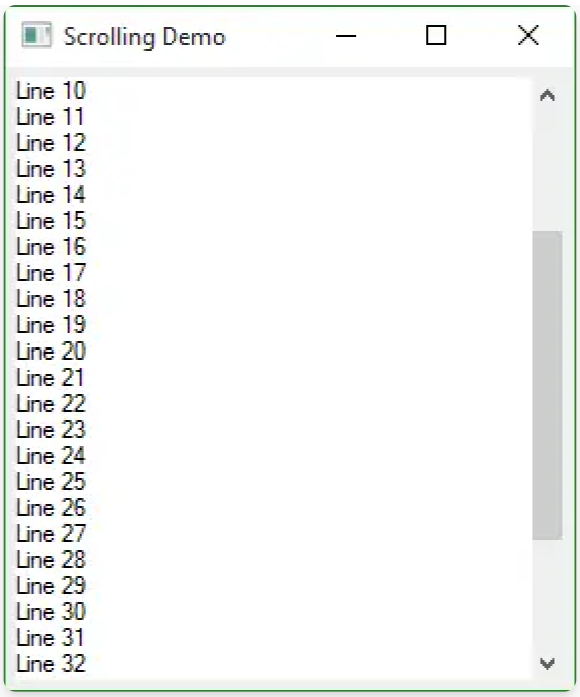
C

```c
static void
CustomHandleMouseWheel(HWND hwnd, int iDelta, BOOL isVertical)
{
    SCROLLINFO si;
    int nPos;
    int nOldPos;

    si.cbSize = sizeof(SCROLLINFO);
    si.fMask = SIF_PAGE | SIF_POS;
    GetScrollInfo(hwnd, (isVertical ? SB_VERT : SB_HORZ), &si);

    // Compute how many lines to scroll.
    nOldPos = si.nPos;
    nPos = nOldPos + WheelScrollLines(pData->hwnd, iDelta, si.nPage,
        // Update the client area.
        ScrollWindowEx(hwnd,
                       (isVertical ? 0 : (nOldPos - nPos) * scrollUnit),
                       (isVertical ? (nOldPos - nPos) * scrollUnit, 0),
                       NULL, NULL, NULL, NULL, SW_ERASE | SW_INVALIDATE);
    }

    static LRESULT CALLBACK
    CustomProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
    {
        switch(uMsg) {
            ...

            case WM_MOUSEWHEEL:
                CustomHandleMouseWheel(hwnd, HIWORD(wParam), TRUE);
                return 0;

            case WM_MOUSEHWHEEL:
                CustomHandleMouseWheel(hwnd, HIWORD(wParam), FALSE);
                return 0;

            ...
        }

        return DefWindowProc(hwnd, uMsg, wParam, lParam);
    }
```

Examples for Download

This time, there are two example projects available for download. You may find links to both of them at the very top of this article.

The simpler allows only vertical scrolling, but otherwise corresponds roughly to all the code provided in the article.

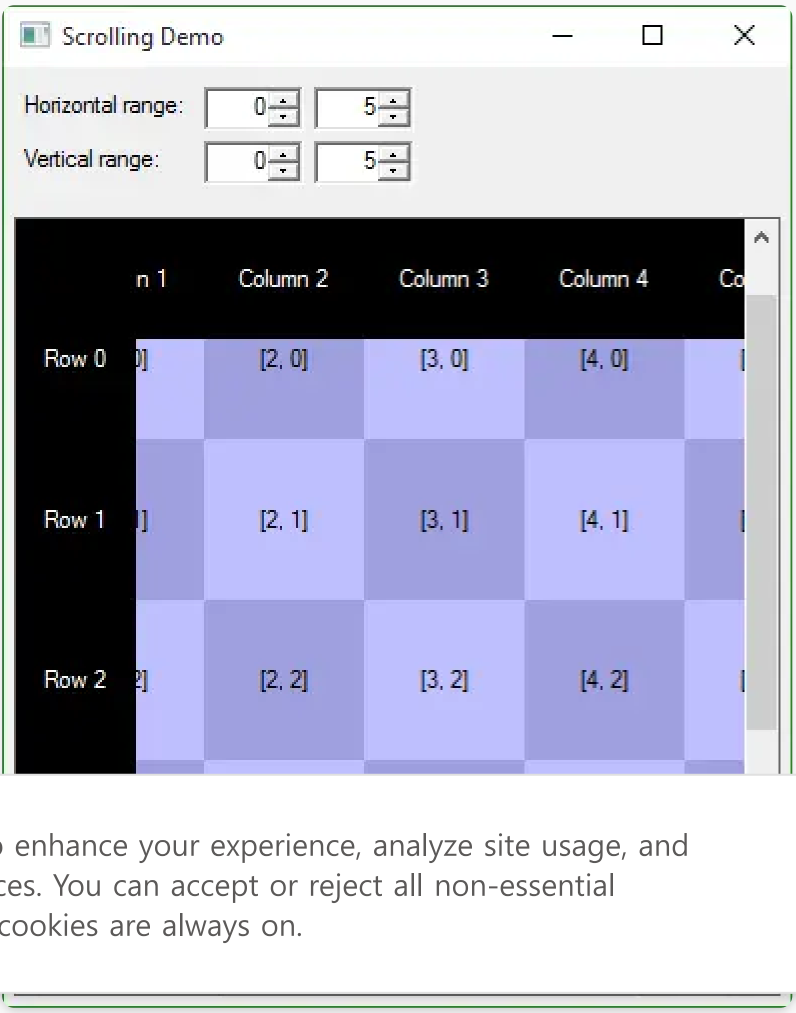


Scrolling Demo

```
Line 10
Line 11
Line 12
Line 13
Line 14
Line 15
Line 16
Line 17
Line 18
Line 19
Line 20
Line 21
Line 22
Line 23
Line 24
Line 25
Line 26
Line 27
Line 28
Line 29
Line 30
Line 31
Line 32
```

Screenshot of the simple demo

The 2nd (and more complex) example does scrolling in vertical as well as horizontal direction, it has a dynamically changing contents so that `nMin` and `nMax` change throughout lifetime of the control, it presents usage of non-trivial scrolling units and last but not least, it shows more advanced use of the function `ScrollWindowEx()` which scrolls only part of the window to keep the headers of columns and rows always visible.



Scrolling Demo

Horizontal range: 0 5
Vertical range: 0 5

	n 1	Column 2	Column 3	Column 4	Co
Row 0	0]	[2, 0]	[3, 0]	[4, 0]	[
Row 1	1]	[2, 1]	[3, 1]	[4, 1]	[
Row 2	2]	[2, 2]	[3, 2]	[4, 2]	[

Screenshot of the more complex demo

Real World Code

In this series, it's already tradition to provide also some links to real-life code demonstrating the topic of the article.

To get better insight, you might find very useful to study the (re)implementation of the scrollbar support in Wine. I especially recommend to pay attention to the function `SCROLL_SetScrollInfo()` which implements core of the `SetScrollInfo()`:

- Scrollbar implementation: https://github.com/mirrors/wine/blob/master/dlls/user32/scroll.c

And, of course, some Wine controls using the implicit scrollbars:

- (Multiline) edit box: https://github.com/mirrors/wine/blob/master/dlls/user32/edit.c
- List view control: https://github.com/mirrors/wine/blob/master/dlls/comctl32/listview.c
- Tree view control: https://github.com/mirrors/wine/blob/master/dlls/comctl32/treeview.c

Finally, also few mCtrl controls using it:

- Grid control: https://github.com/mity/mctrl/blob/master/src/grid.c
- Tree-list view control: https://github.com/mity/mctrl/blob/master/src/treelist.c

Next Time: More About Non-Client Area

Today, we discussed how to implement scrolling support in a control. During the journey, we have lightly touched the topic of non-client area as that's where the scrollbars are living. Next time, we will take a look how to customize a little bit the non-client area, and how to paint in it.

License

by Martin Mitáš
Contributor

90k Views 4.98/ 5

A

Anonymous
2024년 9월 28일

Excellent article. Thank you for posting this.

0 0 Reply

M1

Member 15078716
2022년 7월 9일

Win32

0 0 Reply

A

Anonymous
2021년 4월 29일

Hi Author. Thanks a lot for this project, it is most advanced realization I could found in google. I need for a grid control in my project, and mCtrl contains one which can do almost all what I want. However, reading the docs, can't find a method for total row/column selection. In reference you have written:

CODE

```
With style MC_GS_COMPLEXSEL, any set of cells, even discontinuous
one, may be selected.
```

Das this mean that I miss something, or I should add this functionality myself? Can you explain, please. If it should be added, I will try modify your code, so, be very helpful if will give a short roadmap how to do it by simplest way.. Next problem is more complex, this is a question of styling. I'm developing a *dll with GUI for application which is an external project. This app uses comctl32.dll version 5, and this courses that all control in my *dll also are not styled. All my steps to change this dependency failed. Yes, there is the way to make all controls custom, but is a abnormal amount of work. Is there any kind of legal method to allow my dll, during be loaded by this application, be dependent from comctl32.dll version 6 and use uxtheme.dll? I don't want use any hacks, detour and similar. Thank in advance!

0 0 Reply

Z

zazzacus
2019년 12월 18일

A

Anonymous
2018년 5월 4일

Dear Martin, Thank you for such great tutorial, i learned so much about win32 from your articles! I have a question however, is there any way to compile libmCtrl.lib as static mode without having the need for .dll ? Thank you, William

0 0 Reply

Hide reply

MM

Martin Mitáš AUTHOR
2018년 5월 4일

In short: mCtrl does not really support building as a static lib. Longer answer: I know there were at least two teams/projects who did it. But in both cases it was very hacky. Therefore this feature was never merged into upstream mCtrl project and likely won't ever be. The main technical problem is that static lib cannot reasonably support resources: You would have somehow to merge mCtrl's resource script with resource script of the hosting .EXE/.DLL (and solve potential conflicts in resource IDs between the two) or use only those mCtrl features which do not need to load any resource.

0 0 Reply

A

Anonymous
2018년 5월 4일

Very helpful article!

0 0 Reply

Z

zerhan
2018년 4월 3일

That image (titled "Anatomy of the scrollbar") is worth a thousand words!

0 0 Reply

A

Anonymous
2018년 2월 28일

Great article reminds me a little of the Jeffrey Richters Programming windows book.

0 0 Reply

XTAL256

FM

Franc Morales
2015년 12월 10일

Easy 5. A very complete series.

0 0 Reply

Previous Page 1 of 2 Next