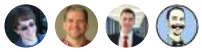
Using the High-Level Input and Output Functions

07/12/2018 • 4 minutes to read • 

> **Important**
>
> This document describes console platform functionality that is no longer a part of our **ecosystem roadmap**. We do not recommend using this content in new products, but we will continue to support existing usages for the indefinite future. Our preferred modern solution focuses on **virtual terminal sequences** for maximum compatibility in cross-platform scenarios. You can find more information about this design decision in our **classic console vs. virtual terminal** document.

The following example uses the high-level console I/O functions for console I/O. For more information about the high-level console I/O functions, see High-Level Console I/O.

The example assumes that the default I/O modes are in effect initially for the first calls to the **ReadFile** and **WriteFile** functions. Then the input mode is changed to turn offline input mode and echo input mode for the second calls to **ReadFile** and **WriteFile**. The **SetConsoleTextAttribute** function is used to set the colors in which subsequently written text will be displayed. Before exiting, the program restores the original console input mode and color attributes.

The example's `NewLine` function is used when line input mode is disabled. It handles carriage returns by moving the cursor position to the first cell of the next row. If the cursor is already in the last row of the console screen buffer, the contents of the console screen buffer are scrolled up one line.

C Copy

```
#include <windows.h>

void NewLine(void);
void ScrollScreenBuffer(HANDLE, INT);

HANDLE hStdout, hStdin;
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;

int main(void)
{
    LPSTR lpszPrompt1 = "Type a line and press Enter, or q to quit: ";
    LPSTR lpszPrompt2 = "Type any key, or q to quit: ";
    CHAR chBuffer[256];
    DWORD cRead, cWritten, fdwMode, fdwOldMode;
    WORD wOldColorAttrs;

    // Get handles to STDIN and STDOUT.

    hStdin = GetStdHandle(STD_INPUT_HANDLE);
```

```
    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if (hStdin == INVALID_HANDLE_VALUE ||
        hStdout == INVALID_HANDLE_VALUE)
    {
        MessageBox(NULL, TEXT("GetStdHandle"), TEXT("Console Error"),
            MB_OK);
        return 1;
    }

    // Save the current text colors.

    if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"),
            TEXT("Console Error"), MB_OK);
        return 1;
    }

    wOldColorAttrs = csbiInfo.wAttributes;

    // Set the text attributes to draw red text on black background.

    if (! SetConsoleTextAttribute(hStdout, FOREGROUND_RED |
            FOREGROUND_INTENSITY))
    {
        MessageBox(NULL, TEXT("SetConsoleTextAttribute"),
            TEXT("Console Error"), MB_OK);
        return 1;
    }

    // Write to STDOUT and read from STDIN by using the default
    // modes. Input is echoed automatically, and ReadFile
    // does not return until a carriage return is typed.
    //
    // The default input modes are line, processed, and echo.
    // The default output modes are processed and wrap at EOL.

    while (1)
    {
        if (! WriteFile(
            hStdout,               // output handle
            lpszPrompt1,           // prompt string
            lstrlenA(lpszPrompt1), // string length
            &cWritten,             // bytes written
            NULL) )                // not overlapped
        {
            MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),
                MB_OK);
            return 1;
        }

        if (! ReadFile(
            hStdin,    // input handle
            chBuffer,  // buffer to read into
            255,       // size of buffer
            &cRead,    // actual bytes read
            NULL) )    // not overlapped
```

```
        break;
    if (chBuffer[0] == 'q') break;
}

// Turn off the line input and echo input modes

if (! GetConsoleMode(hStdin, &fdwOldMode))
{
   MessageBox(NULL, TEXT("GetConsoleMode"), TEXT("Console Error"),
       MB_OK);
   return 1;
}

fdwMode = fdwOldMode &
    ~(ENABLE_LINE_INPUT | ENABLE_ECHO_INPUT);
if (! SetConsoleMode(hStdin, fdwMode))
{
   MessageBox(NULL, TEXT("SetConsoleMode"), TEXT("Console Error"),
       MB_OK);
   return 1;
}

// ReadFile returns when any input is available.
// WriteFile is used to echo input.

NewLine();

while (1)
{
    if (! WriteFile(
        hStdout,               // output handle
        lpszPrompt2,           // prompt string
        lstrlenA(lpszPrompt2), // string length
        &cWritten,             // bytes written
        NULL) )                // not overlapped
    {
        MessageBox(NULL, TEXT("WriteFile"), TEXT("Console Error"),
            MB_OK);
        return 1;
    }

    if (! ReadFile(hStdin, chBuffer, 1, &cRead, NULL))
        break;
    if (chBuffer[0] == '\r')
        NewLine();
    else if (! WriteFile(hStdout, chBuffer, cRead,
        &cWritten, NULL)) break;
    else
        NewLine();
    if (chBuffer[0] == 'q') break;
}

// Restore the original console mode.

SetConsoleMode(hStdin, fdwOldMode);

// Restore the original text colors.
```

```
    SetConsoleTextAttribute(hStdout, wOldColorAttrs);

    return 0;
}

// The NewLine function handles carriage returns when the processed
// input mode is disabled. It gets the current cursor position
// and resets it to the first cell of the next row.

void NewLine(void)
{
    if (! GetConsoleScreenBufferInfo(hStdout, &csbiInfo))
    {
        MessageBox(NULL, TEXT("GetConsoleScreenBufferInfo"),
            TEXT("Console Error"), MB_OK);
        return;
    }

    csbiInfo.dwCursorPosition.X = 0;

    // If it is the last line in the screen buffer, scroll
    // the buffer up.

    if ((csbiInfo.dwSize.Y-1) == csbiInfo.dwCursorPosition.Y)
    {
        ScrollScreenBuffer(hStdout, 1);
    }

    // Otherwise, advance the cursor to the next line.

    else csbiInfo.dwCursorPosition.Y += 1;

    if (! SetConsoleCursorPosition(hStdout,
        csbiInfo.dwCursorPosition))
    {
        MessageBox(NULL, TEXT("SetConsoleCursorPosition"),
            TEXT("Console Error"), MB_OK);
        return;
    }
}

void ScrollScreenBuffer(HANDLE h, INT x)
{
    SMALL_RECT srctScrollRect, srctClipRect;
    CHAR_INFO chiFill;
    COORD coordDest;

    srctScrollRect.Left = 0;
    srctScrollRect.Top = 1;
    srctScrollRect.Right = csbiInfo.dwSize.X - (SHORT)x;
    srctScrollRect.Bottom = csbiInfo.dwSize.Y - (SHORT)x;

    // The destination for the scroll rectangle is one row up.

    coordDest.X = 0;
    coordDest.Y = 0;
```

```
    // The clipping rectangle is the same as the scrolling rectangle.
    // The destination row is left unchanged.

    srctClipRect = srctScrollRect;

    // Set the fill character and attributes.

    chiFill.Attributes = FOREGROUND_RED|FOREGROUND_INTENSITY;
    chiFill.Char.AsciiChar = (char)' ';

    // Scroll up one line.

    ScrollConsoleScreenBuffer(
        h,               // screen buffer handle
        &srctScrollRect, // scrolling rectangle
        &srctClipRect,   // clipping rectangle
        coordDest,       // top left destination cell
        &chiFill);       // fill character and color
}
```

Is this page helpful?

Yes No