

```

// DT3.cpp : Defines the entry point for the application.
//
// Copyright (C) 2001, Intel Corporation
// All rights reserved.
// Permission is hereby granted to merge this program code with other program
// material to create a derivative work. This derivative work may be distributed
// in compiled object form only. Any other publication of this program, in any form,
// without the explicit permission of the copyright holder is prohibited.
//
// Send questions and comments to John.Hyde@intel.com
//
//
// This program is a "minimum feature" USB Dumb Terminal program
// It's goal is to show how a simple serial device can be simply connected to USB using
// a USB I/O device. This example uses the EZ-USB microcontroller and the firmware to
// implement the USB-to-serial-to-USB conversion is also available as an example program.
//
// See the accompanying documentation for a description of this example program
//

#include "stdafx.h"
#include "resource.h"

extern "C" {
// Declare the C libraries used
#include "setupapi.h"
#include "hidsdi.h"
#include "process.h"
}

#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst; // current instance
HWND MainWindow; // Handle for main window
TCHAR szTitle[MAX_LOADSTRING]; // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];
char Buffer[2000]; // A screen full of characters
int BufferIndex; // A pointer into Buffer
const char *SignOn = "\n Select Connect from the File menu to search for 'Serial1' USB I/O device -";
const char *Found = "-> found\n";
const char *Special = " ** F1 Pressed ** ";
HANDLE ReadHandle, WriteHandle;
bool FullDuplex;

// Forward declarations of functions included in this program
ATOM MyRegisterClass(HINSTANCE hInstance);
BOOL InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);
BOOL OpenUSBdevice(char*);
DWORD WINAPI ListenToUSB(LPVOID);
DWORD DisplayError(const char*);

// Entry point for this program
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
// Declare local variables
MSG msg;
HACCEL hAccelTable;

// Initialize global variables
FullDuplex = true; ReadHandle = 0; WriteHandle = 0;
LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
LoadString(hInstance, IDC_DT3, szWindowClass, MAX_LOADSTRING);
MyRegisterClass(hInstance);

// Initialize application
if (!InitInstance (hInstance, nCmdShow)) { return FALSE; }
hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_DT3);

// Main message loop:
while (GetMessage(&msg, NULL, 0, 0)) {
if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
TranslateMessage(&msg);
DispatchMessage(&msg);
}
}
return msg.wParam;
}

```

```

// Register the window class. (required)
ATOM MyRegisterClass(HINSTANCE hInstance) {
    WNDCLASSEX wcex;

    wcex.cbSize      = sizeof(WNDCLASSEX);
    wcex.style       = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = (WNDPROC)WndProc;
    wcex.cbClsExtra  = 0;
    wcex.cbWndExtra  = 0;
    wcex.hInstance   = hInstance;
    wcex.hIcon       = LoadIcon(hInstance, (LPCTSTR)IDI_DT3);
    wcex.hCursor     = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName = (LPCSTR)IDC_DT3;
    wcex.lpszClassName = szWindowClass;
    wcex.hIconSm     = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

    return RegisterClassEx(&wcex);
}

// Initialize the program and create main window
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow) {
    HWND hWnd;

    hInst = hInstance; // Store instance handle in our global variable
    hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
    if (!hWnd) return FALSE;

    MainWindow = hWnd;
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return TRUE;
}

// Process messages for the main window.
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    int ID, Event, Success, i;
    PAINTSTRUCT ps;
    HDC hdc;
    RECT rt;
    HANDLE ThreadHandle;
    HMENU hMainMenu, hSubMenu;
    char WriteBuffer[10];
    DWORD BytesWritten, ThreadID;

    for (i = 0; i<10; i++) WriteBuffer[i] = 0;

    switch (message) {
        case WM_COMMAND:
            ID = LOWORD(wParam); Event = HIWORD(wParam);
            // Parse the menu selections:
            switch (ID) {
                case IDM_ABOUT: DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About); return 0;
                case IDM_EXIT: DestroyWindow(hWnd); return 0;
                case IDM_FULLDUPLEX:
                    // Toggle the Full/Half Duplex flag
                    hMainMenu = GetMenu(hWnd); hSubMenu = GetSubMenu(hMainMenu, 0);
                    if (FullDuplex == !FullDuplex) { CheckMenuItem(hSubMenu, IDM_FULLDUPLEX, MF_BYCOMMAND | MF_CHECKED); }
                    else { CheckMenuItem(hSubMenu, IDM_FULLDUPLEX, MF_BYCOMMAND | MF_UNCHECKED); }
                    return 0;
                case IDM_CONNECT:
                    // Look for the Serial1 USB device
                    if (!OpenUSBdevice("Serial1")) DisplayError("Could not find 'Serial1' device");
                    else {
                        for (i = 0; i<(int)strlen(Found); i++) Buffer[BufferIndex++] = Found[i];
                        InvalidateRect(MainWindow, NULL, 1);
                    }
                    // Serial USB device is open, create a thread to listen for characters
                    ThreadHandle = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ListenToUSB, NULL, 0, &ThreadID);
                    if ((int)ThreadHandle == 0) DisplayError("Could not start USB Listener Thread");
                    // Send an initial character to let the USB I/O device know we are ready
                    Success = WriteFile(WriteHandle, WriteBuffer, 10, &BytesWritten, NULL);
                    DWORD x = GetLastError();
                    if (Success == 0) DisplayError("Could not write to the 'Serial1' device");
            }
            return 0;
    }
}

```

```

        break; // and send unprocessed menu message to Windows
    case WM_PAINT:
        // This is a minimal repaint.
        // A "real" terminal program would process characters such as BS, TAB, CR, LF
        hdc = BeginPaint(hWnd, &ps);
        GetClientRect(hWnd, &rt);
        DrawText(hdc, Buffer, BufferIndex, &rt, 0);
        EndPaint(hWnd, &ps);
        return 0;
    case WM_DESTROY: PostQuitMessage(0); return 0;
    case WM_CHAR:
        // Send the character to the USB I/O device
        WriteBuffer[2] = (char)wParam;
        Success = WriteFile(WriteHandle, WriteBuffer, 10, &BytesWritten, NULL);
        // Create a local echo if required
        if (!FullDuplex) { Buffer[BufferIndex++] = (char)wParam; InvalidateRect(hWnd, NULL, 1); }
        return 0;
    case WM_KEYDOWN:
        // Note that WM_CHAR does not process the Function keys
        if ((char)wParam == 112) { // F1 Function Key, send a string
            for (i=0; i<(int)strlen(Special); i++) {
                WriteBuffer[2] = Special[i];
                Success = WriteFile(WriteHandle, WriteBuffer, 10, &BytesWritten, NULL);
            }
        }
        if ((char)wParam == 113) { // F2 Function key, request a string
            WriteBuffer[2] = (char)wParam;
            Success = WriteFile(WriteHandle, WriteBuffer, 10, &BytesWritten, NULL);
        }
        return 0;
    case WM_CREATE:
        // Say Hello to the User
        for (BufferIndex = 0; BufferIndex<(int)strlen(SignOn); BufferIndex++) Buffer[BufferIndex] =
SignOn[BufferIndex];
        InvalidateRect(hWnd, NULL, 1);
        return 0;
    }
    // Pass unprocessed messages back to Windows
    return DefWindowProc(hWnd, message, wParam, lParam);
}

// Message handler for About box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam) {
    int ID = LOWORD(wParam);
    switch (message) {
        case WM_INITDIALOG: return TRUE;
        case WM_COMMAND: if (ID == IDOK || ID == IDCANCEL) { EndDialog(hDlg, ID); return TRUE; }
        break;
    }
    return FALSE;
}

// Support routines
BOOL OpenUSBdevice(char DeviceName[]) {
    // Search through the attached HID devices for "DeviceName"

    // Declare the local data structures used
    struct _GUID HidGuid;
    SP_INTERFACE_DEVICE_DATA DeviceInterfaceData;
    struct {DWORD cbSize; char DevicePath[256];} FunctionClassDeviceData;
    int Success, HidDevice, i;
    HANDLE PnPHandle, HidHandle;
    unsigned long BytesReturned;
    char buffer[256];
    BOOL Openned;
    SECURITY_ATTRIBUTES SecurityAttributes;

    char ReadBuffer[10];

    // First, get my class identifier and setup the security attributes for Win2000
    HidD_GetHidGuid(&HidGuid);
    SecurityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
    SecurityAttributes.lpSecurityDescriptor = NULL;
    SecurityAttributes.bInheritHandle = false;

    // Get a handle for the Plug and Play node and request currently active HID devices
    PnPHandle = SetupDiGetClassDevs(&HidGuid, 0, 0, 0x12);

```

```

if (int(PnPHandle) == -1) { DisplayError("Could not attach to PnP node"); return FALSE; }

Opened = false;
// Lets look for a maximum of 20 HID devices
for (HidDevice = 0; (HidDevice < 20) && !Opened; HidDevice++) {

// Initialize our Data
DeviceInterfaceData.cbSize = 28; // Length of data structure in bytes

// Is there a HID device at this table entry
Success = SetupDiEnumDeviceInterfaces(PnPHandle, 0, &HidGuid, HidDevice, &DeviceInterfaceData);
if (Success == 1) {
// There is a device here, get it's name
FunctionClassDeviceData.cbSize = 5;
Success = SetupDiGetDeviceInterfaceDetail(PnPHandle, &DeviceInterfaceData,
(PSP_INTERFACE_DEVICE_DETAIL_DATA)&FunctionClassDeviceData, 256, &BytesReturned, 0);
if (Success == 0) {DisplayError("Could not find the system name for this HID device"); return FALSE; }
// Can now open this HID device
HidHandle = CreateFile(FunctionClassDeviceData.DevicePath, GENERIC_READ|GENERIC_WRITE,
FILE_SHARE_READ|FILE_SHARE_WRITE, &SecurityAttributes, OPEN_EXISTING, 0, NULL);
if ((int)HidHandle == -1) { DisplayError("Could not open HID device"); return FALSE; }
// Is it OUR HID device?
if (HidD_GetProductString(HidHandle, buffer, sizeof(buffer))) {
// Compare incoming string with UNICODE string
Opened = true; i = 0;
while (DeviceName[i] != 0) { if (buffer[2*i] != DeviceName[i]) {Opened = false;} i++;}
if (Opened) {
// We have found our device. Open two handles to it (one for each thread)
ReadHandle = CreateFile(FunctionClassDeviceData.DevicePath, GENERIC_READ|GENERIC_WRITE,
FILE_SHARE_READ|FILE_SHARE_WRITE, &SecurityAttributes, OPEN_EXISTING, 0, NULL);
WriteHandle = CreateFile(FunctionClassDeviceData.DevicePath, GENERIC_READ|GENERIC_WRITE,
FILE_SHARE_READ|FILE_SHARE_WRITE, &SecurityAttributes, OPEN_EXISTING, 0, NULL);
}
}
CloseHandle(HidHandle);
} // if (SetupDiEnumDeviceInterfaces . .
} // for (HidDevice = 0; (HidDevice < 20) && !Opened; HidDevice++)
SetupDiDestroyDeviceInfoList(PnPHandle);
return Opened;
}

// Declare the thread that will wait for characters from the USB device
DWORD WINAPI ListenToUSB(LPVOID Param) {
// Collect characters from the USB device and send them to the display
char ReadBuffer[10];
int i, Success;
unsigned long BytesReturned;

while (1) {
Success = ReadFile(ReadHandle, ReadBuffer, 10, &BytesReturned, NULL);
for (i = 2; i<10; i++) if (ReadBuffer[i] != 0) Buffer[BufferIndex++] = ReadBuffer[i];
InvalidateRect(MainWindow, NULL, 1);
};
return 0;
}

// Display various error messages
DWORD DisplayError (const char ErrorText[]) {
int i = MessageBox(MainWindow, ErrorText, "Error", MB_ICONSTOP);
return 0;
}

```