Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Операционные среды и системное программирование

ОТЧЕТ
к лабораторной работе №3
на тему

РЕЕСТР И ЖУРНАЛЫ (WINDOWS). ДОСТУП К РЕЕСТРУ WINDOWS. РАБОТА С ЖУРНАЛАМИ WINDOWS. ДРУГИЕ ВСПОМОГАТЕЛЬНЫЕ СРЕДСТВА УПРАВЛЕНИЯ.

Студент М. А. Шкарубский
Преподаватель Н. Ю. Гриценко

Минск 2023

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Целью выполнения лабораторной работы является разработка приложения для мониторинга, логирования и восстановления уровня звука устройства из реестра.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

2.1 Реестр Windows

Центральная иерархическая база данных, используемая в Windows98, Windows CE, Windows NT и Windows2000, используется для хранения сведений, необходимых для настройки системы для одного или нескольких пользователей, приложений и аппаратных устройств.

Реестр содержит сведения, на которые Windows постоянно ссылается во время операции, такие как профили для каждого пользователя, приложения, установленные на компьютере, типы документов, которые могут создаваться, параметры таблицы свойств для папок и значков приложений, оборудование, которое установлено в системе, и используемые порты [1].

Реестр заменяет большинство текстовых INI-файлов, используемых в файлах конфигурации Windows3.x и MS-DOS, таких как Autoexec.bat и Config.sys. Хотя реестр является общим для нескольких операционных систем Windows, между ними существуют некоторые различия. Куст реестра — это группа ключей, подразделов и значений в реестре с набором вспомогательных файлов, содержащих резервные копии данных. Вспомогательные файлы для всех кустов, кроме HKEY_CURRENT_USER, находятся в папке %SystemRoot%\System32\Config в Windows NT 4.0, Windows 2000, Windows XP, Windows Server 2003 и Windows Vista. Вспомогательные файлы для HKEY_CURRENT_USER находятся в папке %SystemRoot%\Profiles\Username. Расширения имён файлов в этих папках указывают тип содержащихся в них данных. Кроме того, отсутствие расширения иногда может указывать на тип содержащихся в них данных.

2.2 Журнал событий

API журнала событий Windows определяет схему, используемую для написания манифеста инструментирования. Манифест инструментирования идентифицирует поставщика событий и события, которые он регистрирует. Просмотр событий, будет использовать для чтения и отрисовки событий. Чтобы записать события, определённые в манифесте, можно использовать функции, включённые в API трассировки событий (ETW) [2].

Журнал событий Windows заменяет API ведения журнала событий, начиная с операционной системы Windows Vista.

Приложения и библиотеки используют манифест инструментирования для определения поставщиков инструментирования и событий, которые они записывают. Манифест является XML-файлом, содержащим элементы, определяющие поставщика. Соглашение заключается в том, чтобы использовать .man в качестве расширения для манифеста. Манифест должен соответствовать XSD манифесту события.

События можно использовать из каналов или из файлов журнала. Для использования событий можно использовать все события или указать выражение XPath, определяющее события, которые вы хотите использовать. Сведения об определении элементов и атрибутов события, которые можно использовать в выражении XPath. Для запросов используются следующие функции:

1 EvtQuery: функция запроса события. Можно указать порядок, в котором возвращаются события (от старых к новым (по умолчанию) или от новых к старым), а также указать, следует ли допускать неправильно сформированные выражения XPath в запросе (дополнительные сведения о том, как функция игнорирует неправильно сформированные выражения.

2 EvtGetQueryInfo: функция для определения успешных и неудачных запросов. Если не передать флаг EvtQueryTolerateQueryErrors, функция EvtQuery завершится ошибкой с первой ошибкой, найденной в запросе. Если запрос завершается сбоем с ERROR_EVT_INVALID_QUERY.

3 EvtNext: функция перечисляет события в результирующем наборе если вызвать её в цикле, пока она не вернёт false, а функция GetLastError ERROR_NO_MORE_ITEMS.

3 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ

В результате выполнения лабораторной работы было спроектировано и написано приложение, использующее их для логирования истории изменения громкости устройства и восстановления последнего сохраненного уровня громкости устройства.

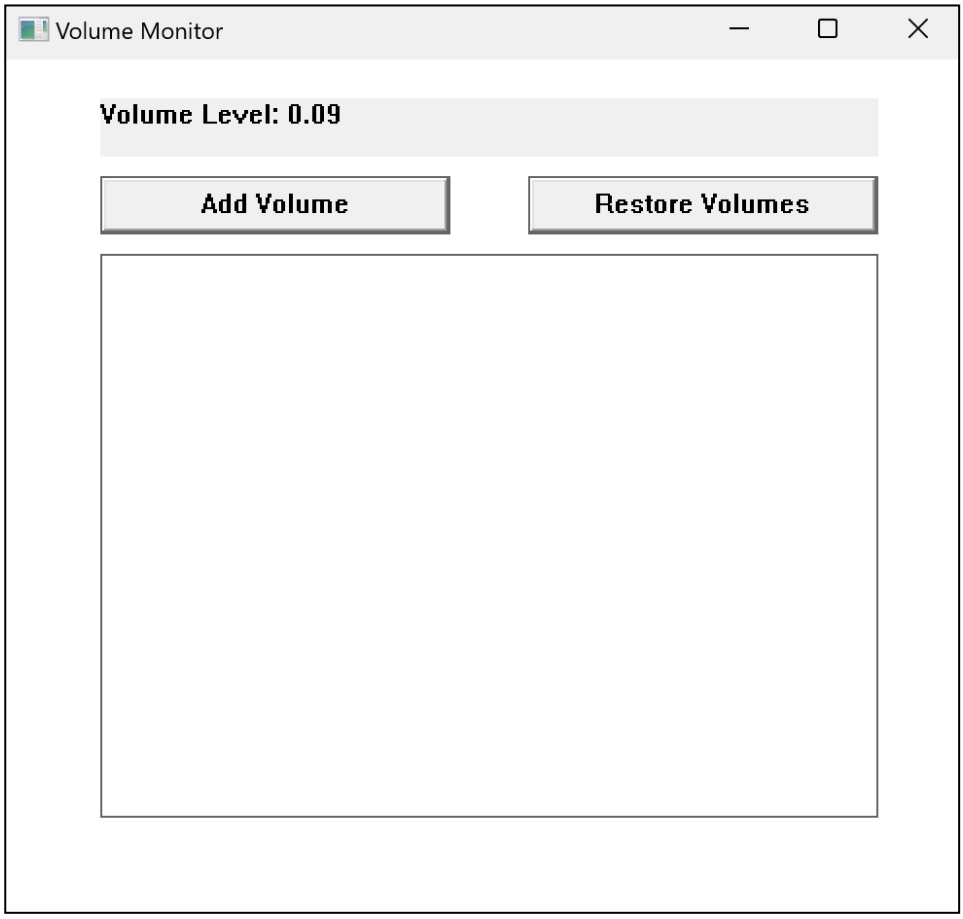

Рисунок 1 – Окно приложения

В процессе использования приложения пользователь может логировать (сохранять) уровни громкости системы в текущий момент, просматривать историю изменений громкости устройства (см. рисунок 2) и восстанавливать громкость устройства с последнего сохраненного уровня (см. рисунок 3).

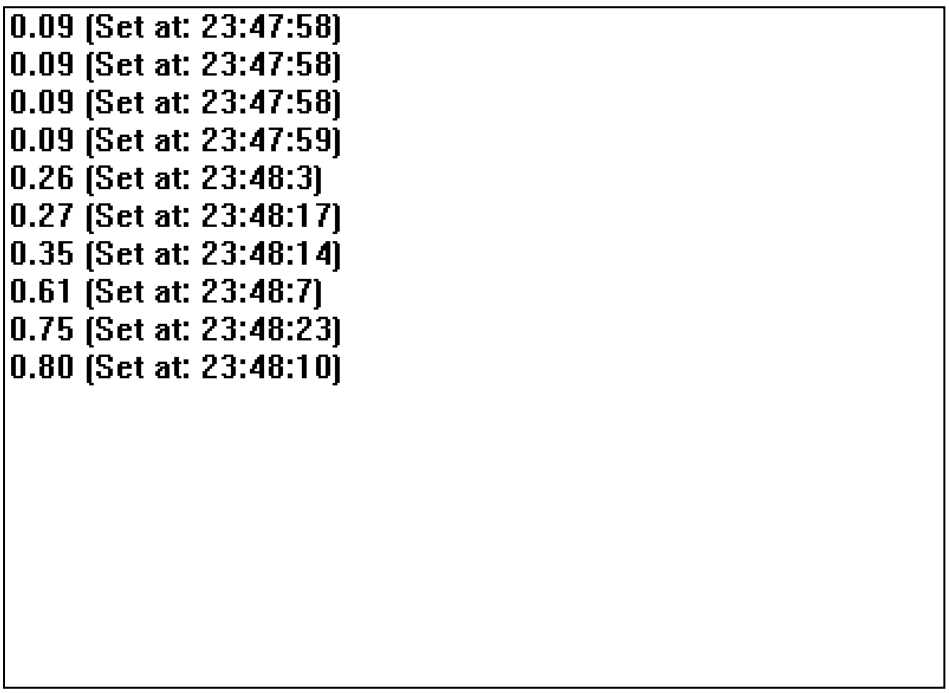

Рисунок 2 – Список изменения уровней громкости системы

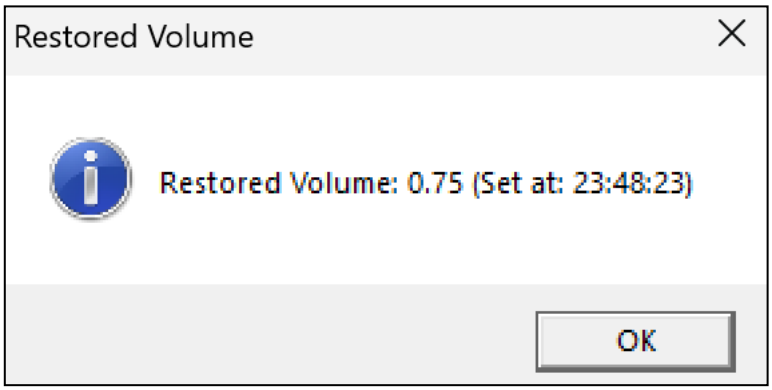

Рисунок 3 – Уведомление о восстановлении предыдущей громкости

ЗАКЛЮЧЕНИЕ

В результате выполнения лабораторной работы были изучены реестр Windows (Windows Registry), а также другие вспомогательные средства управления. Было спроектировано и написано приложение, использующее их для логирования истории изменения громкости устройства и восстановления последнего сохраненного уровня громкости устройства.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Сведения о реестре Windows для опытных пользователей [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/troubleshoot/windows-server/performance/windows-registry-advanced-users.

[2] Синхронизация процессов Win32 API [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/sync/interprocess-synchronization.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

Листинг 1 – Файл Lab5.cpp

```
#include <Windows.h>
#include <CommCtrl.h>
#include <Mmdeviceapi.h>
#include <Endpointvolume.h>
#include <wrl.h>
#include <wrl/client.h>
#include <iostream>
#include <vector>

using namespace Microsoft::WRL;

std::vector<std::pair<float, SYSTEMTIME>> volumeList;
IAudioEndpointVolume* endpointVolume = nullptr;
HWND g_hWnd;
HWND g_hTextLabel;
HWND g_hAddButton;
HWND g_hRestoreButton;
HWND g_hListBox;

void UpdateListBox() {
    SendMessage(g_hListBox, LB_RESETCONTENT, 0, 0); // Clear the ListBox

    for (const auto& entry : volumeList) {
        SYSTEMTIME st = entry.second;
        wchar_t volumeText[100];
        swprintf_s(volumeText, L"%.2f (Set at: %d:%d:%d)", entry.first,
st.wHour, st.wMinute, st.wSecond);
        SendMessage(g_hListBox, LB_ADDSTRING, 0, (LPARAM)volumeText); // Add
the entry to the ListBox
    }
}

// Function to update the volume text
void UpdateVolumeText(float volume) {
    wchar_t volumeText[100];
    swprintf_s(volumeText, L"Volume Level: %.2f", volume);
    SetWindowText(g_hTextLabel, volumeText);
}

// Function to add the current volume and timestamp to the Registry
void AddVolumeToRegistry(float volume) {
    HKEY hKey;
    if (RegCreateKeyEx(HKEY_CURRENT_USER, L"Software\\VolumeChangeMonitor",
0, NULL, 0, KEY_WRITE, NULL, &hKey, NULL) == ERROR_SUCCESS) {
        SYSTEMTIME st;
        GetSystemTime(&st);
        FILETIME ft;
        SystemTimeToFileTime(&st, &ft);
        RegSetValueEx(hKey, L"Volume", 0, REG_BINARY, (BYTE*)&volume,
sizeof(volume));
```

```
        RegSetValueEx(hKey, L"Timestamp", 0, REG_BINARY, (BYTE*)&ft,
sizeof(ft));
        RegCloseKey(hKey);
    }

    SYSTEMTIME st;
    GetSystemTime(&st);
    volumeList.emplace_back(volume, st);

    UpdateListBox();
}

// Function to restore the volume list from the Registry
void RestoreVolumesFromRegistry() {
    HKEY hKey;
    if (RegOpenKeyEx(HKEY_CURRENT_USER, L"Software\\VolumeChangeMonitor", 0,
KEY_QUERY_VALUE, &hKey) == ERROR_SUCCESS) {
        float volume;
        FILETIME ft;
        DWORD dataSize = sizeof(volume);
        if (RegQueryValueEx(hKey, L"Volume", NULL, NULL, (BYTE*)&volume,
&dataSize) == ERROR_SUCCESS) {
            dataSize = sizeof(ft);
            if (RegQueryValueEx(hKey, L"Timestamp", NULL, NULL, (BYTE*)&ft,
&dataSize) == ERROR_SUCCESS) {
                SYSTEMTIME st;
                FileTimeToSystemTime(&ft, &st);
                wchar_t volumeText[100];
                swprintf_s(volumeText, L"Restored Volume: %.2f (Set at: %d:
%d:%d)", volume, st.wHour, st.wMinute, st.wSecond);
                MessageBox(g_hWnd, volumeText, L"Restored Volume",
MB_ICONINFORMATION);

                HRESULT hr = endpointVolume-
>SetMasterVolumeLevelScalar(volume, NULL);

                if (SUCCEEDED(hr))
                    UpdateListBox();
            }
        }
        RegCloseKey(hKey);
    }

    UpdateListBox();
}

DWORD WINAPI VolumeChangeThread(LPVOID) {
    float currentVolume;
    HRESULT hr;

    while (true) {
        hr = endpointVolume->GetMasterVolumeLevelScalar(&currentVolume);
        if (SUCCEEDED(hr)) {
            UpdateVolumeText(currentVolume); // Update the volume text on the
window

            // Check for volume changes and add them to the Registry
            float newVolume;
```

```
            hr = endpointVolume->GetMasterVolumeLevelScalar(&newVolume);

            if (SUCCEEDED(hr) && newVolume != currentVolume) {
                // Volume has changed
                UpdateVolumeText(newVolume);
                currentVolume = newVolume;

                // Add the volume and timestamp to the Registry
                // AddVolumeToRegistry(newVolume);
            }
        }
        Sleep(10); // Check volume every second (adjust as needed)
    }

    return 0;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam) {
    switch (message) {
    case WM_CREATE:
    {
        // Initialize COM
        CoInitialize(NULL);

        // Create a label to display volume text
        g_hTextLabel = CreateWindow(L"STATIC", L"", WS_CHILD | WS_VISIBLE,
50, 20, 400, 30, hWnd, NULL, NULL, NULL);
        SetWindowText(g_hTextLabel, L"Volume Level: 0.00");

        // Create an "Add Volume" button
        g_hAddButton = CreateWindow(L"BUTTON", L"Add Volume", WS_CHILD |
WS_VISIBLE | BS_DEFPUSHBUTTON, 50, 60, 180, 30, hWnd, (HMENU)1, NULL, NULL);

        // Create a "Restore Volumes" button
        g_hRestoreButton = CreateWindow(L"BUTTON", L"Restore Volumes",
WS_CHILD | WS_VISIBLE | BS_DEFPUSHBUTTON, 270, 60, 180, 30, hWnd, (HMENU)2,
NULL, NULL);

        // Create a ListBox to display volume entries
        g_hListBox = CreateWindow(WC_LISTBOX, L"", WS_CHILD | WS_VISIBLE |
WS_BORDER | LBS_STANDARD, 50, 100, 400, 300, hWnd, NULL, NULL, NULL);

        // Initialize the audio endpoint volume interface
        IMMDeviceEnumerator* deviceEnumerator = nullptr;
        IMMDevice* defaultPlaybackDevice = nullptr;
        HRESULT hr = CoCreateInstance(__uuidof(MMDeviceEnumerator), NULL,
CLSCTX_INPROC_SERVER, __uuidof(IMMDeviceEnumerator),
(void**)&deviceEnumerator);

        if (SUCCEEDED(hr)) {
            hr = deviceEnumerator->GetDefaultAudioEndpoint(eRender, eConsole,
&defaultPlaybackDevice);
        }

        if (SUCCEEDED(hr)) {
            hr = defaultPlaybackDevice-
>Activate(__uuidof(IAudioEndpointVolume), CLSCTX_INPROC_SERVER, NULL,
```

```
(void**)&endpointVolume);
        }

        if (deviceEnumerator) {
            deviceEnumerator->Release();
        }
        if (defaultPlaybackDevice) {
            defaultPlaybackDevice->Release();
        }
        if (FAILED(hr)) {
            // Handle the error
            MessageBox(hWnd, L"Failed to initialize audio interface.",
L"Error", MB_ICONERROR);
        }
    }

        // Start a separate thread to monitor volume changes
        CreateThread(nullptr, 0, VolumeChangeThread, nullptr, 0, nullptr);
        break;

    case WM_COMMAND:
    {
        if (LOWORD(wParam) == 1) {
            // "Add Volume" button clicked
            float currentVolume;
            if (SUCCEEDED(endpointVolume-
>GetMasterVolumeLevelScalar(&currentVolume))) {
                // Add the current volume to the Registry
                AddVolumeToRegistry(currentVolume);
                // MessageBox(hWnd, L"Volume added to the list.", L"Add
Volume", MB_ICONINFORMATION);
            }
        }
        else if (LOWORD(wParam) == 2) {
            // "Restore Volumes" button clicked
            RestoreVolumesFromRegistry();
        }
    }
        break;

    case WM_CLOSE:
        // Release the audio endpoint volume interface and uninitialize COM
        if (endpointVolume) {
            endpointVolume->Release();
            endpointVolume = nullptr;
        }
        CoUninitialize();
        PostQuitMessage(0);
        break;

    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nCmdShow) {
```

```
    // Register the window class
    WNDCLASSEX wc = { sizeof(WNDCLASSEX), CS_CLASSDC, WndProc, 0L, 0L,
GetModuleHandle(NULL), NULL, NULL, NULL, NULL, L"VolumeChangeMonitor",
NULL };
    RegisterClassEx(&wc);

    // Create the window
    g_hWnd = CreateWindow(wc.lpszClassName, L"Volume Monitor",
WS_OVERLAPPEDWINDOW, 100, 100, 510, 480, NULL, NULL, wc.hInstance, NULL);

    // Show the window
    ShowWindow(g_hWnd, nCmdShow);
    UpdateWindow(g_hWnd);

    // Message loop
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}
```