Le génie pour l'industrie

GHL Poke Machine Report

Presented to
Prof. Pascal GIARD

Prepared by
Gabriel Ouaknine-Beaulieu
Gabriel.Ouaknine-Beaulieu.1@ens.etsmtl.ca
École de technologie supérieure

Montréal, May 26, 2022

Contents

Summary

Guitar Hero is a music rhythm game with guitar-like remotes originally published by RedOctane in 2005 and now published by Activision. This franchise has been published on many consoles like the Xbox360, the PlayStation 3 (PS3), the PlayStation 4 (PS4), the Wii, the Wii U. Computer users can play Clone Hero which is a Guitar Hero like game. Actually, the Clone Hero community has at least 30k members. The main motivation of this project is to give to the guitar hero community a way to easily use guitar hero remotes on Windows computers. The project is focused on PS3, PS4 and Wii U six frets guitars.

On top of helping thousands of people enjoy this rhythm game, this project is interesting in another way. Since the game has not been published on newer consoles and only on vintage one, the guitar's hero remotes are harder to find. With a way to use the remotes on the computer, it gives a way to avoid the obsolescence of those remotes. Fighting the obsolescence of any product is an excellent way to fight climate changes. If this project can help other projects to use vintage remotes on computers, it can be beneficial for the environment and the consumers.

A proper driver for Linux users has already been published but there's no proper way to use the remotes on a Windows computer. An attempt has been made to emulate an Xbox 360 controller with the PS3 / Wii U / iOS guitars. However, it requires a convoluted installation and it does not fully benefit from the windows Human Interface Devices (HID) drivers. This project gives a robust and a simple solution to use PS3, PS4 and Wii U guitar dongles on Windows computers. As found in the Linux driver project, the dongles need a poking mechanism to avoid falling in a standby mode. In the standby mode, the dongle does not recognize the use of several buttons simultaneously which is essential to play.

The solution described in this report is an Microsoft Foundation Class (MFC) desktop application that pokes the dongles at regular intervals. The Windows HID driver takes care of all the human interactions. The program operates with a thread for each dongle found.

The result is good. It is a simple-to-use solution. The level of abstraction gives a robustness. Because of it, the application should work on future versions of Windows. The application is already available publicly here [1]. I have great hope that this program will be popular amongst the Clone Hero community.

Chapter 1

Introduction

The rhythm game Clone Hero uses guitar-like remotes. The community didn't have an easy way to use guitar hero live dongles on Windows computers. To solve this problem, I created GHLPokeMachine as a project for an internship. In this report, I will describe my project. Firstly, the background needed to understand the project will be explained. Then, the application code and functionality will be discussed. Finally, the final results will be shown.

Chapter 2

Background

The intent of this section is to give enough technical knowledge to understand the project. The topics covered are: the guitar remote, the Universal Serial Bus (USB) protocol, USB development on Windows, the human interfaces devices and globally unique identifiers.

2.1 Dongles and Hardwares

The remote has two parts: a dongle and a guitar as seen in Figure 2.2. The dongle is connected to the console or in our case the computer and the guitar is used by the player to play. The fret buttons are pressed while the strum bar is struck to play notes like a real guitar. The goal of the game is to play the note with a good tempo as seen in Figure 2.1. The GHTV Button is used to pause the game. The whammy bar is used to add tremolo to the notes.

Figure 2.1: Clone Hero Gameplay [2]

To synchronize the guitar, you need to power the guitar. The status LEDs will blink. Then, you should press the power button again, the status LEDs will link fast for 30s. While the status LED blinks fast, press the dongle connect button. Once synchronized, all the status LEDs will stop blinking.

Figure 2.2: Dongle and Guitar adapted from [3] [4]

1. Fret Buttons
2. Strum Bar
3. Whammy Bar
4. Status LED
5. GHTV Button
6. Directional pad/Power Button
7. Dongle Connect Button
8. Dongle Status LED

2.2 Universal Serial Bus

The universal serial bus is an industry standard for connectors, communication protocol and else. It was created in 1994 by a consortium of seven companies (Compaq, DEC, IBM, Intel, Microsoft, NEC and Northern Telecom). The goal was to ease the data transfer, develop a universal connection and ease the use of peripherals with a "plug and play" mechanic[5]. Needless to say, the USB was a hit and it's now very popular. Nowadays, everyone knows what a USB connector looks like. It's

important to understand that USB devices work with a “plug and play” mechanic for the plug and play section.

2.3 Plug and Play

The Plug and Play (PnP) driver is there to ease the life of the users. It’s responsible for the automatic and dynamic recognition of installed hardware, the hardware resource allocation, the loading of appropriate drivers, the programming interface for drivers to interact with the PnP system and the mechanisms for drivers and applications to learn of changes in the hardware environment and take appropriate actions[6].

The way it works on Windows is when a device is plugged, a driver called plug and play will automatically detect the newly plugged device. Then, it will look in a list of drivers already installed on the computer to find the appropriate driver for the device. If there’s no appropriate driver, Windows will automatically try to install an appropriate driver by downloading it, installing it from the device or else. Then the driver will have the appropriate behavior for the device[5].

To select the appropriate driver, all drivers associated with a device are ranked. Windows assigns the best scores to drivers with a trusted signature. Then it assigns the second best score to drivers installed by an INF DDInstall section that has an .nt platform extension. Then it assigns the third score to drivers installed with an .nt platform extension. Then it will assign the worst signature score[7].

Creating a driver that would be directly referenced by the PnP driver would be really good. Since Windows assigns the best scores to drivers with a trusted signature, it is hard to create a driver without a trusted signature. Also, it is convoluted to use a driver that doesn’t have a trusted signature. That’s why the project is an application. Still, the PnP driver connect the HID driver to the dongle.

2.4 USB development on Windows

The USB development on Windows can be done with different frameworks. As seen in Figure 2.3, the three main frameworks are the user mode framework, the User Mode Driver Framework (UMDF) and the Kernel Mode Driver Framework (KMDF)[8]. Each of those frameworks operate with a different level of abstraction. The user mode framework is used in the applications. It will allow the application to communicate with the different drivers. The UMDF and the KMDF are used in drivers. Both work in a really similar way, but their virtual address spaces are different. In the kernel mode, there’s only one virtual address space shared by all drivers and the kernel. It allows a driver to access all virtual addresses. Therefore, a driver in this mode can accidentally write to the wrong virtual address and crash the system. On the other hand, all user mode applications have their own private virtual address spaces and private handles. In this mode, the drivers cannot accidentally write in a wrong virtual address. If the application crashes, only this application crashes[9]. Since creating drivers are convoluted, GHLPokeMachine is an application in the user mode.

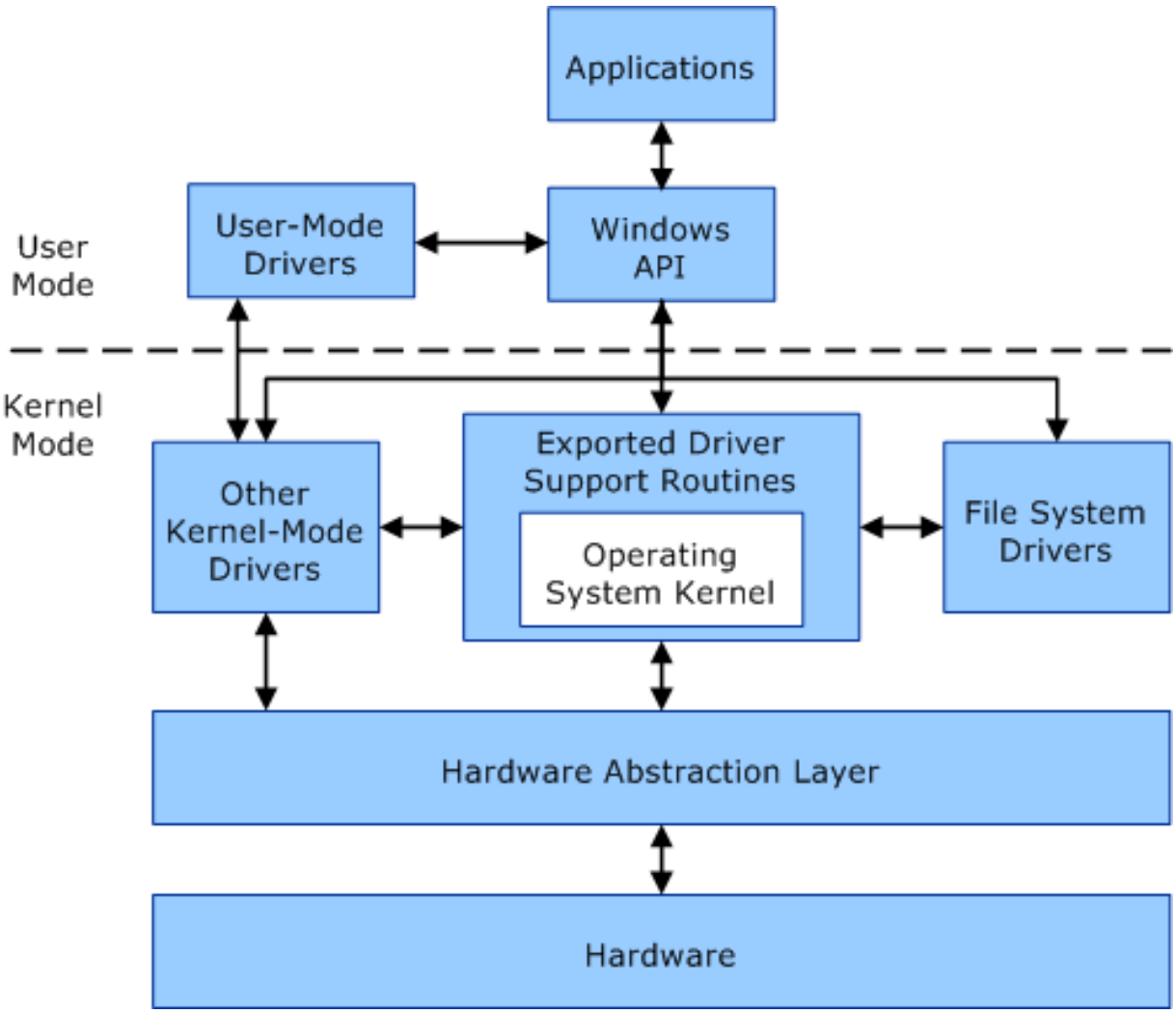

Figure 2.3: communication between user-mode and kernel-mode components [9]

Drivers written with the KMDF or the UMDF are really similar in structure. The drivers have different routines that are being executed when called. Both need to have at least an entry and an exit routine. Once started by the PnP, the driver will start by executing the entry routine. The entry routine should identify the other routines. Then, the driver will receive I/O Request Package (IRP) structures. "The IRP structure is a partially opaque structure that represents an I/O request packet"[10]. With those IRPs, the driver will start the appropriate routine. The driver will stop when it receives the IRP asking for the exit routine[11] [12] [13] [14]. If GHLPokeMachine was a driver, that's how it would have work. An application communicating with a driver send IRPs. It's a predefined structure. It is useful to know that we cannot change the structure sent to the drivers through an application.

A USB device can provide information about itself through structures called USB descriptors. There're different kinds of useful descriptors. There's the device descriptor, the configuration descriptor, the interface descriptor, the endpoint descriptor. The device descriptor contains relevant information about the device. This descriptor contains the Vendor IDentificator (VID) and the

Product IDentificator (PID) fields. These fields are unique to the device and the vendor. These fields are usually used to identify the device[15]. With the VID and the PID, the device will be identified in the find devices paths section.

2.5 Globally Unique IDentifier

There are different classes of drivers. The class that will interest us mainly will be the HID class since the guitar hero remote is a HID class device. Every class has an associated class Globally Unique IDentifier (GUID). For example the HID device interface class GUID is 4D1E55B2 - F16F - 11CF - 88CB - 001111000030[16]. The goal of those GUID is to provide a number so large that it will be very unlikely that two GUID will be the same. There's 2^{128} possibilities [17]. In comparison, Population Reference Bureau estimates that by 2050 about 113 billion ($2^{36.72}$) people will have lived on Earth [18]. The device interface class GUID is used in the definitions section and in the find devices paths section.

2.6 Human Interface Devices

Human interfaces devices are devices such as keyboards, mice, game controllers and nowadays alphanumeric displays, bar code readers, volume controls, etc. Built on top of the USB protocol, there's the HID protocol for such devices with an associated driver. Before HID driver class was created, creating specialized driver was often required for each device. The HID devices use reports to communicate. There's three type of reports. The input report is data sent from the HID device to the application. The output report is data sent from the application to the device. The feature report is data that can be manually read and written, typically related to configuration information [19]. Microsoft has already a library to send report. To send an output report, the `HidD_SetOutputReport` function can be used. To send feature report, the `HidD_SetFeature` function can be used [20]. To send the poking message, GHLPokeMachine send output reports in the section send poking message.

Chapter 3

GHLPokeMachine

Through reverse engineering efforts and as shown by the HID linux driver's for Activision GH Live PS3, Wii U, and PS4 Guitar devices project, the dongle need a poking message to stay active [21]. Our application is divided in a front end, a back end and threads like in Figure 3.1. The goal of the back end is to find the devices paths on the computer, start a thread for each devices. Then, the threads send a poking message. The application is also be able to stop properly. The application is based on a MFC application project to ease the interface development.

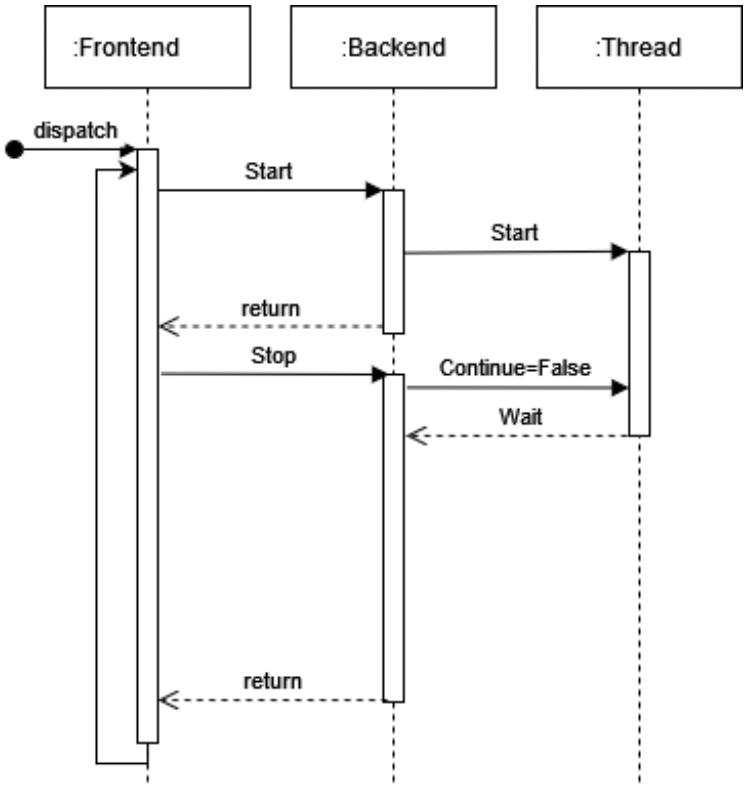

Figure 3.1: Sequence diagram of GHLPokeMachine

3.1 Definitions

In the header file displayed in Figure 3.2, the VID and the PID of the supported devices are defined. Those VID and PID are used to identify which define are used by the application. Each of those device need specific poking message which are also defined in the header file along with a sleeping time. The sleeping time is the period interval to send the poking message. A structure is defined to send the needed data in the thread functions.

```
#define ONE_SECOND              ( 1000 ) // millisecond
#define POKE_MESSAGE_LENGTH     ( 9 )
#define MAX_THREADS ( 4 )

#define PS3_WIIU_VID_PID        ( L"vid_12ba&pid_074b" )
#define PS3_WIIU_POKE_MESSAGE   { 0x02, 0x02, 0x08, 0x20, 0x00, 0x00, 0x00, 0x00, 0x00 }
#define PS3_WIIU_SLEEP_TIME     ( 10 * ONE_SECOND ) // 10 seconds

#define PS4_VID_PID             ( L"vid_1430&pid_07bb" )
#define PS4_POKE_MESSAGE        { 0x30, 0x02, 0x08, 0x0A, 0x00, 0x00, 0x00, 0x00, 0x00 }
#define PS4_SLEEP_TIME          ( 8 * ONE_SECOND ) // 8 seconds

//GUID for HID Devices
DEFINE_GUID(GUID_DEVINTERFACE_HID,  0x4D1E55B2,
                                    0xF16F, 0x11CF,
                                    0x88, 0xCB, 0x00, 0x11,
                                    0x11, 0x00, 0x00, 0x30);

typedef enum DeviceType { UNKNOWN_DEVICE = 0, PS3_WIIU, PS4 };

//structure to send the needed data in the thread
typedef struct _DEVICE_DATA
{

    BOOL                    HandlesOpen;
    HANDLE                  DeviceHandle;
    WINUSB_INTERFACE_HANDLE InterfaceHandle;
    int                     SleepTime;
    CWnd                    *DlgItem;
    TCHAR                   DevicePath[MAX_PATH];
    CHAR                    PokeMessage[POKE_MESSAGE_LENGTH];
} DEVICE_DATA, * PDEVICE_DATA;
...
```

Figure 3.2: Header File Definitions

3.2 Find devices paths

Figure 3.3 show the function `StartGHPoke`. This function is there to find the devices paths and start a thread for each devices. The function follow the example of `SetupDiGetClassDevsW` function to find the devices paths [22]. Firstly, it enumerate all the devices connected on the computer that are related to HID class GUID (Figure 3.3).

```
// Enumerate all devices exposing the HID interface
    deviceInfo = SetupDiGetClassDevs(   &GUID_DEVINTERFACE_HID,
                                        NULL,
                                        NULL,
                                        DIGCF_PRESENT  | DIGCF_DEVICEINTERFACE);
    if (deviceInfo == INVALID_HANDLE_VALUE)
    {
        hr = HRESULT_FROM_WIN32(GetLastError());
        return hr;
    }
    interfaceData.cbSize = sizeof(SP_DEVICE_INTERFACE_DATA);
    ...
```

Figure 3.3: `StartGHPoke` function : Enumerate all devices

Then, through a while loop it can check every devices (Figure 3.4). The `SetupDiEnumDeviceInterfaces` is used to load the interface data. Interface data is a structure used to find the detail data structure which contain the device path. Depending on the loaded device, the memory space needed for the detail data can vary. To find the required length, the `SetupDiGetDeviceInterfaceDetail` function is used twice. To check if the device is an appropriate dongle, the function `CheckVidPid` is used.

```
while (Continue)
{
    // Get the next interface (index i) in the result set
    bResult = SetupDiEnumDeviceInterfaces(  deviceInfo,
                                            NULL,
                                            &GUID_DEVINTERFACE_HID,
                                            i,
                                            &interfaceData);
    if (FALSE == bResult){...}

    // Get the size of the path string
    // We expect to get a failure with insufficient buffer
    bResult = SetupDiGetDeviceInterfaceDetail(  deviceInfo,
                                                &interfaceData,
                                                NULL,
                                                0,
                                                &requiredLength,
                                                NULL);
    if (FALSE == bResult && ERROR_INSUFFICIENT_BUFFER != GetLastError()){...}

    // Allocate temporary space for SetupDi structure
    detailData = (PSP_DEVICE_INTERFACE_DETAIL_DATA)LocalAlloc(LMEM_FIXED, requiredLength);
    if (NULL == detailData){...}
    detailData->cbSize = sizeof(SP_DEVICE_INTERFACE_DETAIL_DATA);
    length = requiredLength;

    // Get the interface's path string
    bResult = SetupDiGetDeviceInterfaceDetail(  deviceInfo,
                                                &interfaceData,
                                                detailData,
                                                length,
                                                &requiredLength,
                                                NULL);
    if (FALSE == bResult){...}

    /* Check if The device is one from the list */
    if (Device = CheckVidPid(detailData->DevicePath)){...}
    /* Increment Device Interface Index*/
    i++;
}
...
```

Figure 3.4: `StartGHPoke` function : Enumerate all devices loop

In the `CheckVidPid` function (Figure 3.5), the `wcsstr` function is used to find if the VID and the PID is in the device path. Windows put the VID and the PID in the device path. The function return the device type to initialize the `DEVICE_DATA` structure properly.

```
DeviceType CheckVidPid(WCHAR *DevicePath)
{
    if (wcsstr(DevicePath, PS4_VID_PID))
        return PS4;
    else if (wcsstr(DevicePath, PS3_WIIU_VID_PID))
        return PS3_WIIU;
    else
        /* Not a recognized Device*/
        return UNKNOWN_DEVICE;
}
```

Figure 3.5: `CheckVidPid` function

To summarize like on the left part of the Figure 3.6, all devices are listed. Then the specific details of the devices are found to obtain the devices path. In the devices path, there's the VID and the PID which are used to identify if it's a appropriate device.

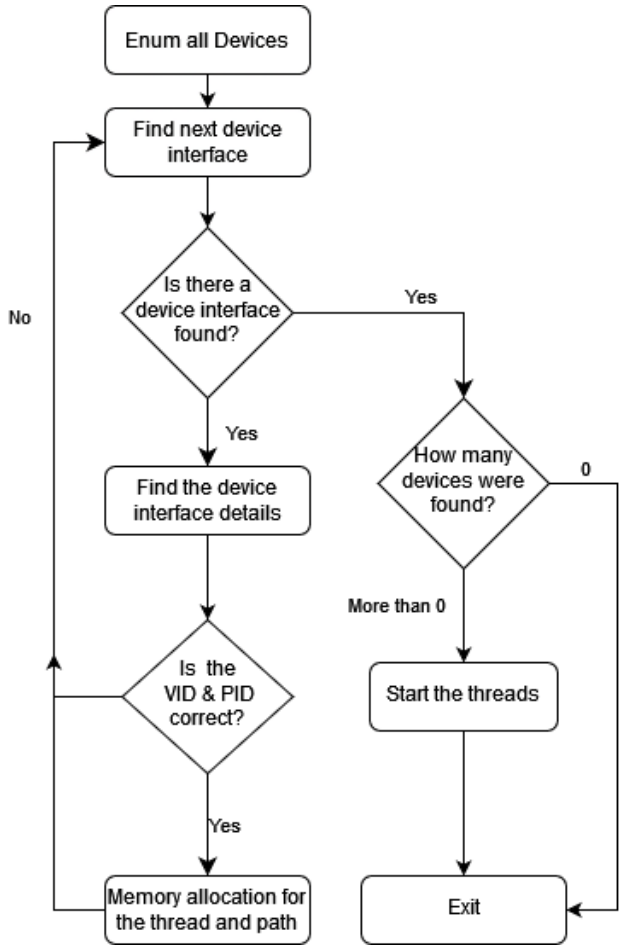

Figure 3.6: Find device path flow chart

3.3 Start a thread

Once the appropriate device is found, some memory is allocated for the device data structure and the thread (Figure 3.7). The memory address can change through memory reallocation when there's multiple dongle. To avoid accessing bad memory emplacement, the thread are started once all devices are found.

```
/* Check if The device is one from the list */
if (Device = CheckVidPid(detailData->DevicePath))
{
    /* Memory Allocation for data Pointer */
    *pDeviceData = (PDEVICE_DATA)realloc(*pDeviceData, (j + 1) * sizeof(DEVICE_DATA));
    if (*pDeviceData == NULL){...}

    /* Initialize Data */
    if (InitializeData( &(*pDeviceData)[j],
                        detailData->DevicePath, Device))
    {

        /* Memory Allocation for data Pointer */
         *hThreadArray = (PHANDLE)realloc(  *hThreadArray,
                                            (j + 1) * sizeof(HANDLE));
         if (*hThreadArray == NULL){...}

        /* Memory Allocation for data Pointer */
        *dwThreadIdArray = (PDWORD)realloc( *dwThreadIdArray,
                                            (j + 1) * sizeof(DWORD));
        if (*dwThreadIdArray == NULL){...}
        j++;
    }
}
```

Figure 3.7: `StartGHPoke` function : Memory Allocation

In the `InitializeData` function (Figure 3.8), the device path is open with the `createfile` function. The poking message and the sleeping time are set using the `DeviceType found` in the `CheckVidPid` function. A poking message is sent to make sure everything work. Some devices dongle have more than one interface and only one is used to send report. The poking message is sent through an output report.

```
BOOL InitializeData(    PDEVICE_DATA pDeviceData,
                        WCHAR* DevicePath,
                        DeviceType Device)
{
    /* Copy Device Path in device Data */
    if (StringCbCopy(   pDeviceData->DevicePath,
                        MAX_PATH - 1, DevicePath) != S_OK){...}
    /* Open File */
    pDeviceData->DeviceHandle = CreateFile(pDeviceData->DevicePath,
                                GENERIC_WRITE | GENERIC_READ,
                                FILE_SHARE_WRITE | FILE_SHARE_READ,
                                NULL,
                                OPEN_EXISTING,
                                FILE_FLAG_OVERLAPPED,
                                NULL);
    if (INVALID_HANDLE_VALUE == pDeviceData->DeviceHandle){...}

    /* Flag for handle open */
    pDeviceData->HandlesOpen = TRUE;

    /* Initialize Specific Device Data */
    switch (Device)
    {
    case PS4:
        memcpy_s(   pDeviceData->PokeMessage,
                    POKE_MESSAGE_LENGTH,
                    Ps4PokeMessage,
                    POKE_MESSAGE_LENGTH);
        pDeviceData->SleepTime = PS4_SLEEP_TIME;
        break;
    case PS3_WIIU:
        memcpy_s(   pDeviceData->PokeMessage,
                    POKE_MESSAGE_LENGTH,
                    Ps3WiiuPokeMessage,
                    POKE_MESSAGE_LENGTH);
        pDeviceData->SleepTime = PS3_WIIU_SLEEP_TIME;
        break;
    default:
        /* Not supported */
        break;
    };

    /* Send a first Poke. Some device have more than one interface */
    if (!HidD_SetOutputReport(  pDeviceData->DeviceHandle,
                                pDeviceData->PokeMessage,
                                POKE_MESSAGE_LENGTH * sizeof(CHAR))
        ){...}

    return TRUE;
}
```

Figure 3.8: `InitializeData` function

Once all devices in the list are checked, `SetupDiEnumDeviceInterfaces` function will return `ERROR_NO_MORE_ITEMS`. If no devices were found, an error will be return; else, the thread are started with the `CreateThread` function (Figure 3.9).

```
bResult = SetupDiEnumDeviceInterfaces(  deviceInfo,
                                        NULL,
                                        &GUID_DEVINTERFACE_HID,
                                        i,
                                        &interfaceData);
if (FALSE == bResult)
{
    // We would see this error if no devices were found
    if (ERROR_NO_MORE_ITEMS == GetLastError() && NULL != FailureDeviceNotFound)
    {
        if (j == 0)
        {
            *FailureDeviceNotFound = TRUE;
        }
        else
        {
            for (int i = 0; i < j; i++)
            {
                (*hThreadArray)[i] = CreateThread(
                    NULL,                   // default security attributes
                    0,                      // use default stack size
                    SendPokeMessage,        // thread function name
                    &(*pDeviceData)[i],     // argument to thread function
                    0,                      // use default creation flags
                    &(*dwThreadIdArray)[i]);// returns the thread identifier
                if ((*hThreadArray)[i] == NULL){...}
            }
            *FailureDeviceNotFound = FALSE;
            *pNbRemote = j;
        }
    }
    hr = HRESULT_FROM_WIN32(GetLastError());
    SetupDiDestroyDeviceInfoList(deviceInfo);
    return hr;
}
```

Figure 3.9: `StartGHPoke` function : Start the threads

Essentially, once an appropriate device is found, a data structure is initialized with the poking message, the handle to send the reports etc. A report is sent to make sure everything is correct. Then, there's a memory allocation for a thread that will be started once every device in the list has been checked. The right side Figure 3.6 represent this sequence.

3.4 Send poking message

The threads execute the `SendPokeMessage` function (Figure 3.10). The function wait until the start signal has been given, then it will write in the interface which kind of dongle is active. A while loop is active until the continue variable is set to false. The thread will send the poke via an output report. If sending the report has failed, it will give an error message. If everything is correct, it will sleep until for a sleeping interval and start again. Once the continue variable is set to false, the thread will close the device handle. This process is shown in Figure 3.11.

```
DWORD WINAPI SendPokeMessage(LPVOID lpParam)
{
    HRESULT hr = S_OK;
    PDEVICE_DATA DeviceData = (PDEVICE_DATA)lpParam;
    BOOL    bResult;
    int i = 0;

    while ( !Start || !Continue)
    {
        Sleep(ONE_SECOND);
    }
    //Write in the interface which dongle it is
    SetStaticText(DeviceData->DlgItem, GetDeviceString(DeviceData));

    while (Continue)
    {
        /* Send Poke */
        bResult = HidD_SetOutputReport( DeviceData->DeviceHandle,
                                        DeviceData->PokeMessage,
                                        POKE_MESSAGE_LENGTH * sizeof(CHAR));
        if (!bResult)
        {
            /* If error eg: Deconnexion */
            hr = GetLastError();
            //Send a message
            MessageBox(NULL, (LPCWSTR)L"One device was disconnected\n"
            "Make sure your USB dongle is plugged into the computer,"
            "press stop and then start",
            (LPCWSTR)GetDeviceString(DeviceData),
            MB_ICONEXCLAMATION | MB_OK | MB_DEFBUTTON1);
            //Write in the interface that there's a connection problem
            SetStaticText(DeviceData->DlgItem, L"Connection Problem");

            CloseHandle(DeviceData->DeviceHandle);
            DeviceData->HandlesOpen = FALSE;
            return hr;
        }
        /* Wait for next poke */
        Sleep(DeviceData->SleepTime);
    }
    CloseHandle(DeviceData->DeviceHandle);
    DeviceData->HandlesOpen = FALSE;
    return hr;
}
```

Figure 3.10: `SendPokeMessage` function

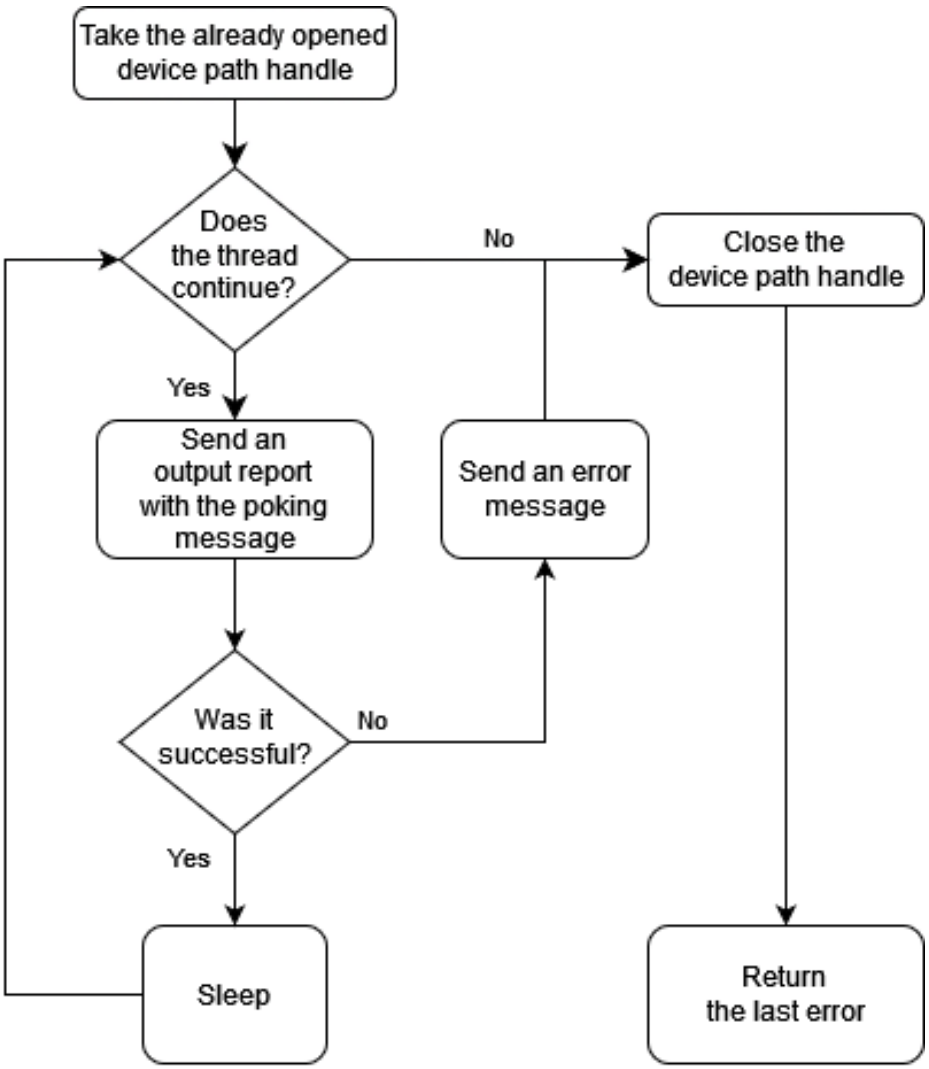

Figure 3.11: Flow chart of the threads

3.5 Stop properly

To stop all the threads, the `StopGHPoke` function (Figure 3.12) is called. It set the Continue variable to false, therefore stopping the while loop of the different threads. Then it will wait until all thread finished with the `WaitForMultipleObjects` function. It will free all the dynamic memory allocation and reset all variables to be able to start the application again.

```
HRESULT StopGHPoke( PDWORD   *dwThreadIdArray,
                    PHANDLE  *hThreadArray,
                    PDEVICE_DATA *pDeviceData )
{
    Continue =  FALSE;

    WaitForMultipleObjects(j, *hThreadArray, TRUE, INFINITE);

    if (*dwThreadIdArray != NULL)
    {
        free(*dwThreadIdArray);
        *dwThreadIdArray = NULL;
    }
    if (*hThreadArray != NULL)
    {
        free(*hThreadArray);
        *hThreadArray = NULL;
    }

    if (*pDeviceData != NULL)
    {
        free(*pDeviceData);
        *pDeviceData = NULL;
    }

    SetStartThread(FALSE);
    Continue = TRUE;

    j = 0;

    return 0;
}
```

Figure 3.12: `StopGHPoke` function

Chapter 4

Results

The goal of this section is to show the application and the USB communication with and without the application running.

4.1 How it look

I tried to do a minimalist design. The application is shown when it doesn't find devices on the left side of the Figure 4.1. On the right side of the Figure 4.1, the application is shown running. To start looking for device, the user needs to click on the start button. Once it found devices, all connected devices will be listed. Then, to stop the application, the user can click on the stop button. Pressing the exit button will close the application.

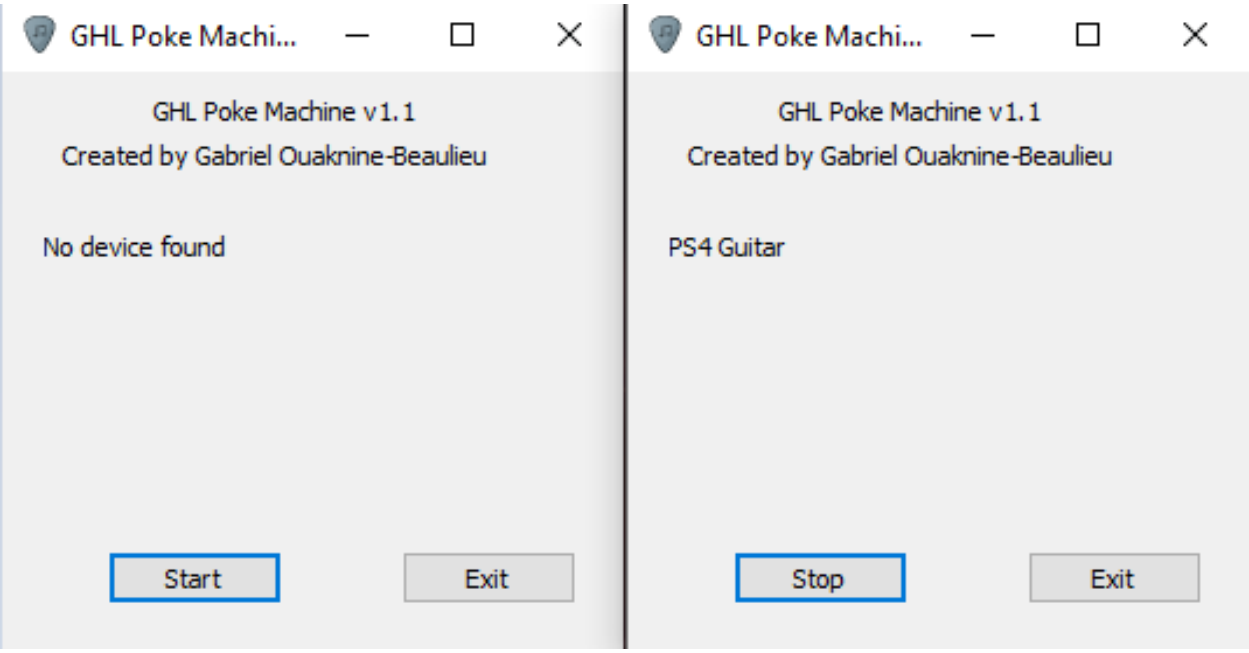

Figure 4.1: Screenshot of GHLPokeMachine Application

4.2 USB Communications

In the Figure 4.2, the communication between the computer and the dongle are shown. To collect this data, I used a USB sniffer named Beagle USB 480 Protocol Analyzer. It spies on the different packets sent in a USB connection. The first communications are exchanges of the address and the descriptors for the drivers to be able to work properly. Then, the dongle will send input reports carrying the information of the frets buttons state and the strum bar state. As said before, without the application running, the dongle will not send the frets buttons state and the strum bar state together.

Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
33	0:04.551.354	33.0 ms				[34 SOF]	[Frames: 661 - 694]
34	0:04.584.328	0 B		00	00	Set Address	Address=11
43	0:04.585.358	9.00 ms				[10 SOF]	[Frames: 695 - 704]
44	0:04.594.578	18 B		11	00	Get Device Descriptor	Index=0 Length=18
63	0:04.595.360	4.00 ms				[5 SOF]	[Frames: 705 - 709]
64	0:04.599.148	41 B		11	00	Get Configuration Descriptor	Index=0 Length=255
83	0:04.599.702	4 B		11	00	Get String Descriptor	Index=0 Length=255
102	0:04.599.973	26 B		11	00	Get String Descriptor	Index=2 Length=255
121	0:04.600.360	2.08 us				[1 SOF]	[Frame: 710]
122	0:04.600.450	0 B		11	00	Control Transfer (STALL)	Index=0 Length=10
131	0:04.601.360	75.0 ms				[76 SOF]	[Frames: 711 - 786]
132	0:04.676.646	18 B		11	00	Get Device Descriptor	Index=0 Length=18
151	0:04.676.991	9 B		11	00	Get Configuration Descriptor	Index=0 Length=9
178	0:04.677.370	2.08 us				[1 SOF]	[Frame: 787]
179	0:04.677.398	41 B		11	00	Get Configuration Descriptor	Index=0 Length=41
198	0:04.678.049	0 B		11	00	Set Configuration	Configuration=1
208	0:04.678.370	1.00 ms				[2 SOF]	[Frames: 788 - 789]
209	0:04.679.517	0 B		11	00	Set Idle	Duration=Indefinite Report=0
219	0:04.680.370	1.00 ms				[2 SOF]	[Frames: 790 - 791]
220	0:04.679.946	160 B		11	00	Get Report Descriptor	Index=0 Length=224
249	0:04.682.371	4.00 ms				[5 SOF]	[Frames: 792 - 796]
250	0:04.686.379	64 B		11	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]
251	0:04.686.379	64 B		11	01	IN txn	01 80 80 80 80 08 00 00 00 00 00
255	0:04.687.371	143 ms				[144 SOF]	[Frames: 797 - 940]
256	0:04.690.375	64 B		11	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]

Figure 4.2: USB Communication without the application

Once the application is started, it will send output reports with the appropriate poking message. In the Figure 4.3, a PS4 is connected. The output packet highlighted in blue is the same as mentioned in Figure 3.2. With the output report, the dongle will send the frets buttons state and the strum bar state together.

Index	m:s.ms.us	Len	Err	Dev	Ep	Record	Summary
429	0:08.103.935	71.0 ms				[72 SOF]	[Frames: 1957 - 2028]
430	0:08.175.340	9 B		09	00	Set Output Report [48]	Length=9
431	0:08.175.340	8 B		09	00	SETUP txn	21 09 30 02 00 00 09 00
435	0:08.175.365	9 B		09	00	OUT txn [1 POLL]	30 02 08 0A 00 00 00 00 00
444	0:08.175.418	0 B		09	00	IN txn [6 POLL]	
449	0:08.175.944	79.0 ms				[80 SOF]	[Frames: 2029 - 60]
450	0:08.106.938	64 B		09	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]
456	0:08.255.954	147 ms				[148 SOF]	[Frames: 61 - 208]
457	0:08.258.958	64 B		09	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]
463	0:08.403.973	151 ms				[152 SOF]	[Frames: 209 - 360]
464	0:08.406.977	64 B		09	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]
470	0:08.555.992	151 ms				[152 SOF]	[Frames: 361 - 512]
471	0:08.558.999	64 B		09	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]
477	0:08.708.012	151 ms				[152 SOF]	[Frames: 513 - 664]
478	0:08.711.015	64 B		09	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]
484	0:08.860.031	151 ms				[152 SOF]	[Frames: 665 - 816]
485	0:08.863.037	64 B		09	01	Input Report [1]	Axes=[X:128 Y:128 Z:128 Rz:128 Rx:0...]

Figure 4.3: USB Communication with the application

Chapter 5

Conclusion

In conclusion, this project is a Windows application to help the Clone Hero community and other Guitar Hero like games fans. It gives them an easy-to-use solution to be able to play PS3, PS4 and Wii U six frets guitars on Windows computer. Nothing existed to be able to play PS4 six frets guitars on Windows computer prior to the application. It is increasing in popularity and have helped about one hundred persons only a month after its release.The application is released under a general public license and can be downloaded here [1]. GHLPokeMachine, like its name state, work with a secret poking mechanism that keeps the dongles from falling in a standby mode. It is an application that is robust and as future proof as it can be because it is based on Microsoft Foundation Class (MFC) libraries and the HID API. It should work on the future Windows versions as Microsoft developers put effort in keeping their libraries backward compatible. This project is complete. Yet, the Clone Hero community is still missing a way to play on Xbox One six frets guitar dongles on Windows computer. It would be a great addition to this project.

Acknowledgement

This project wouldn't have been possible without the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

I'm also thankful for the financial support, the help and the time given by my mentor, Professor Pascal Giard. He guided me through this project, helped me in the design and all sort of other details. It is a real pleasure to work with him. I'm looking forward for future collaboration

Finally, I would like to thanks everyone in the guitar hero like community that tested my application.

Bibliography

[1] G. Ouaknine-Beaulieu, *GHLPokeMachine*, Apr. 2022. [Online]. Available: https://github.com/Octave13/GHLPokeMachine (visited on 04/10/2022).

[2] neil9000, *Clone Hero (Images)*, LaunchBox. [Online]. Available: https://gamesdb.launchbox-app.com/games/images/93459 (visited on 04/10/2022).

[3] *Guitar Hero Live Ps4 Dongle Replacement*, Epay. [Online]. Available: https://www.zppays.gq/ProductDetail.aspx?iid=75963388&pr=84.88 (visited on 04/10/2022).

[4] M. Walton, *Guitar Hero Live review: This is how to make rhythm games relevant again*, ARS Technica, Oct. 2015. [Online]. Available: https://arstechnica.com/gaming/2015/10/guitar-hero-live-review-this-is-how-you-make-rhythm-games-relevant-again/ (visited on 04/10/2022).

[5] M. Kuh, *La norme USB en quelques mots*, Sep. 2002. [Online]. Available: http://u.s.b.free.fr/pdf/L_USB_et_sa_norme_v1.pdf (visited on 04/10/2022).

[6] T. Hudek and T. Sherer, *Introduction to Plug and Play*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-plug-and-play (visited on 04/10/2022).

[7] T. Hudek and Mamont, *Signature Score*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/signature-score--windows-vista-and-later- (visited on 04/10/2022).

[8] M. Hopkins and V. Surgos, *Getting started with USB development*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-concepts-for-all-developers (visited on 04/10/2022).

[9] A. Viviano, *User mode and kernel mode*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode (visited on 04/10/2022).

[10] *IRP structure (wdm.h)*, Microsoft, Feb. 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_irp (visited on 04/10/2022).

[11] M. Hopkins, B. Lattimer, E. Graff, and N. Schonning, *Understanding the USB client driver code structure (UMDF)*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/understanding-the-umdf-template-code-for-usb (visited on 04/10/2022).

[12] M. Hopkins, K. Sharkey, A. Viviano, and B. Lattimer, *Understanding the USB client driver code structure (KMDF)*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/understanding-the-kmdf-template-code-for-usb (visited on 04/10/2022).

[13] P. Langester, *Windows Driver Development Tutorial for Beginners*, Jan. 2018. [Online]. Available: https://www.youtube.com/playlist?list=PLZ4EgN7ZCzJyUT-FmgHsW4e9BxfP-VMuo (visited on 04/10/2022).

[14] ——, *Windows Kernel Programming Tutorials for Beginners*, Jul. 2017. [Online]. Available: https://www.youtube.com/playlist?list=PLZ4EgN7ZCzJx2DRXTRUXRrB2njWnx1kA2 (visited on 04/10/2022).

[15] M. Hopkins and A. Viviano, *Standard USB descriptors*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/usbcon/standard-usb-descriptors (visited on 04/10/2022).

[16] D. R. Ted Hudek, *GUID_DEVINTERFACE_HID*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/guid-devinterface-hid (visited on 04/10/2022).

[17] K. Azad, *The Quick Guide to GUIDs*. [Online]. Available: https://betterexplained.com/articles/the-quick-guide-to-guids/ (visited on 04/10/2022).

[18] T. Kaneda and C. Haub, *How Many People Have Ever Lived on Earth?* Population Reference Bureau, Jan. 2020. [Online]. Available: https://scorecard.prb.org/howmanypeoplehaveeverlivedoneart (visited on 04/10/2022).

[19] M. Hopkins, B. Lattimer, and E. Graff, *Introduction to Human Interface Devices (HID)*, Microsoft, Mar. 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/ (visited on 04/10/2022).

[20] M. Hopkins and B. Lattimer, *Sending HID reports*, Microsoft, Dec. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/sending-hid-reports (visited on 04/10/2022).

[21] D. Nguyen, *HID driver for Activision GH Live PS3, Wii U, and PS4 Guitar devices*, Jun. 2021. [Online]. Available: https://github.com/dynamix1337/hid-ghlive-dkms (visited on 04/10/2022).

[22] *SetupDiGetClassDevsW function (setupapi.h)*, Microsoft, Oct. 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/api/setupapi/nf-setupapi-setupdigetclassdevsw (visited on 04/10/2022).