KeySim

MCU PART
API PART

Ataberk Öklü
METU

Table of Contents

Motivation

The project consists of three elements, namely; CPU - the PC side, MCU, the controller for the device with input keys, and the GUI / API for assigning proper actions to the keys of the device. The device keys are determined to be digital input to the MCU, processing and transmitting the dedicated function indicator to the CPU via serial communication. To establish the communication between MCU and PC, UART is selected.

Environment

STM32 ARM cortex MCUs are planned for controlling the input device. STM32CubeMX and KEIL µVision are used to set the MCU and program. API / GUI is designed using C# and Visual Studio software. STM32F103C8 MCU is used for this project.

MCU as HID device

In this project, MCU must behave as Human Interface Device to properly function for keyboard and mouse actions. The intended project contains 15+ key for DEL, CTRL, HOME, PAGE DOWN, PAGE UP, SHIFT and mouse actions like right click and left click, and so on.

MCU Requirements

Firstly, our HID interface is though USB connection, therefore, the selected MCU should support USB interface. This communication requires 48Mhz clock, this should be satisfied in order to establish proper communication between MCU and PC- CPU. USB clock signal is very restricted in terms of clock frequency, and should be strictly obeyed. Additionally, this configuration should be done via High Speed Clock (HSE) with external ceramic oscillator or crystal. The quality of the clock source important to satisfy the strict clock frequency requirement. In the use of STM32L476 Nucleo-64 Discovery Board, you may need an external crystal oscillator to use HSE clock.

Secondly, of course you might need a cable to connect the MCU to PC, from USB D+ and USB D-pins. Also, to use Human Interface Device option, you need to select Full Speed USB Device mode for MCU via STM32CUbeMX.

Finally, you may need tools to program and debug your MCU. For STM32 MCUs, ST-Link tool and software can be used. Also, you are able to upload your built code by using KEIL µVision software, after installing ST-Link software.

First Step: Configuring the MCU

STM32CubeMX software ease the configuration process of the MCU. We need an indicator or HW debugger LED to investigate the state of program, a bunch of button to trigger specified actions and interrupts, an UART interface to communicate with CPU to set button functions, and USB Device interface for converting button inputs to computer actions.

Clock Configuration

Clock, firstly, should satisfy the requirements of USB interface, mentioned in MCU Requirements section, above. USB HID needs **48 MHz HSE** clock signal. In the STM32CubeMX, HSE in the RCC options must be set to "**Crystal / Ceramic Resonator**" option, as shown in the Figure 1 below.
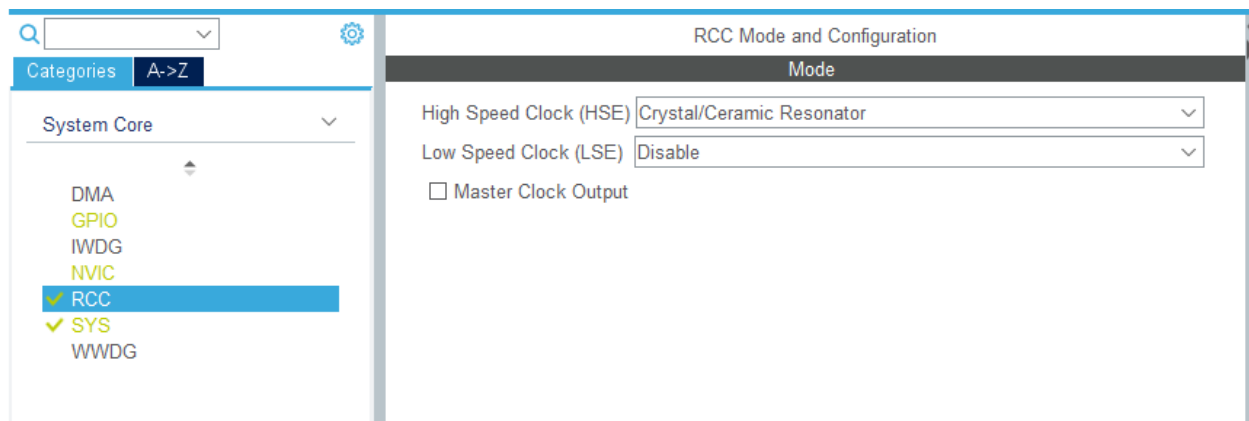
Figure 1 – RCC configuration / STM32CubeMX / STM32F103C8

After this setting, you may encounter a clock problem in the "**Clock Configuration**" section in the STM32CubeMX user interface. The first must is having **48 MHz HSE** clock signal in the USB Clock side. You can either select auto solve option or set manually, eventually, configuration should resemble to configuration shown in Figure 2, below.
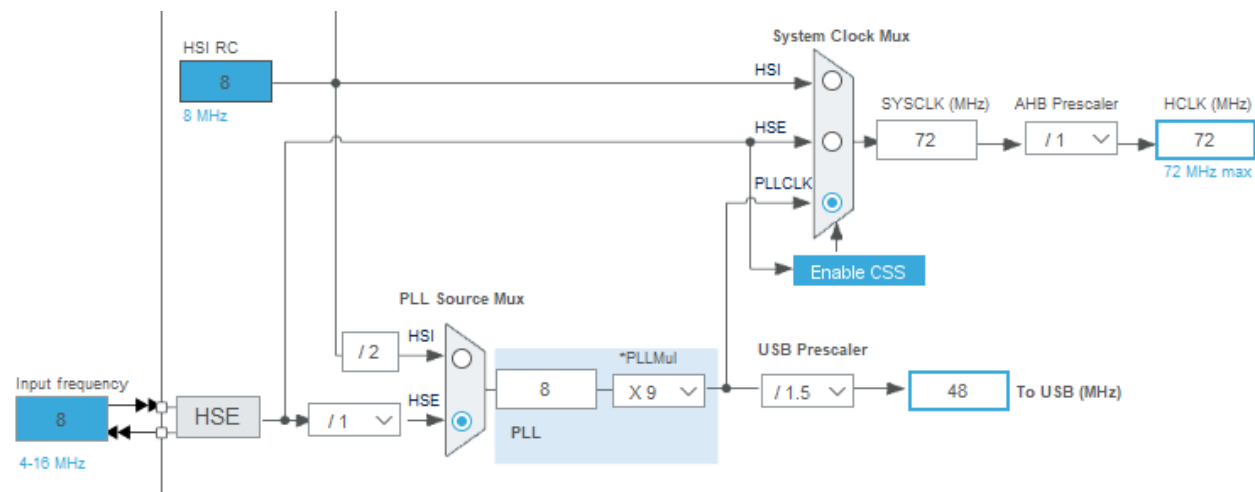
Figure 2 – USB HSE Clock Configuration / STM32CubeMX / STM32F103C8

USB HID Configuration

In order to have and USB HID device, first we need to active the USB communication under "**Connectivity**" menu in the STM32CubeMX interface. Make sure that both "**Device (FS)**" selected and "**Speed**" set to "**Full Speed 12Mbit/s**" option. In other MCUs, you may encounter different options like "**USB mode**", "**Activate SOF**" or Activate "**Activate NOE**". Some of the MCUs may require Start of Frame connection to obtain 48 MHz HSE clock signal, this information can be found in product datasheet. If your MCU can work as both host and device, for this purpose please use only device mode. After configuration, USB menu should be like shown in the Figure 3, below.
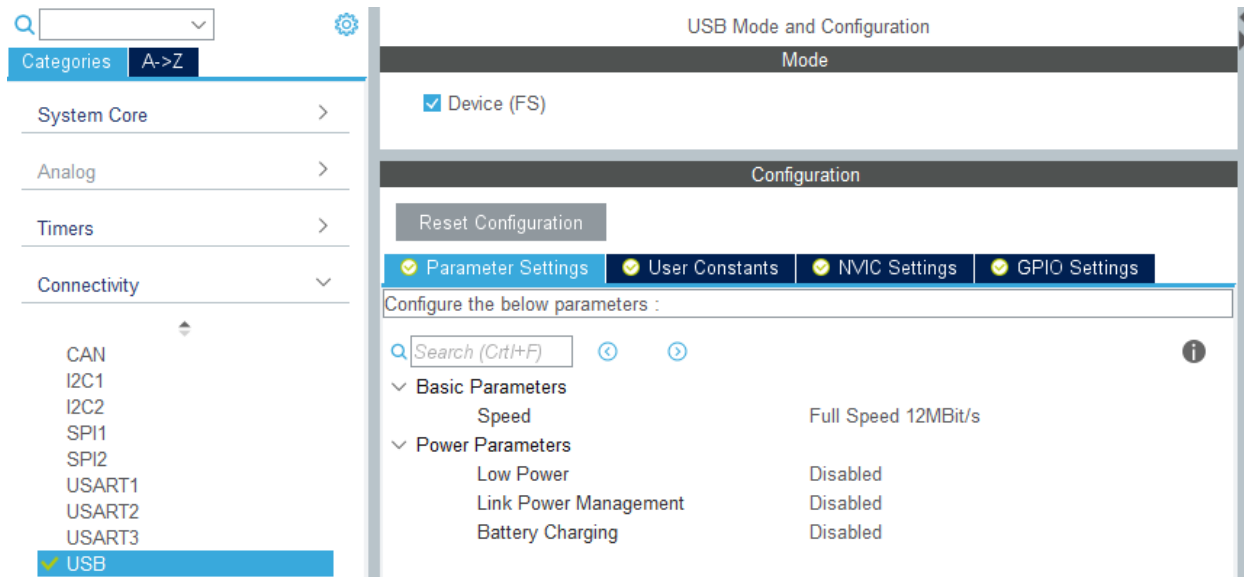
Figure 3 – USB Connection Configuration

This set allow us to configure the device behavior in USB interface. In order to achieve special key activation for PC, Human Interface Device option should be selected. Under "**Middleware**" menu "**USB_DEVICE**" option should be activated, and should be set. If the case that other USB modes are exist, all should be "not accessible" but the "**USB_DEVICE**" option. In this menu, firstly, "**Class for FS IP**" setting is set to "**Custom HID Class**", because, this project contains both pointer (Mouse) reports and keyboard reports at the same time. The most important parameter is the "**USBD_CUSTOM_HID_REPORT_DESC_SIZE**". This parameter contains the information of how many bytes you have in your report description. Report Description is going to be elaborated in next section. Since we are using both keyboard and mouse descriptions, our description size is **101 bytes**. The other important parameter is the "**USBD_MAX_NUM_INTERFACES**", related with the number of different kind of input types. In our case, this is 2, one for mouse and other for keyboard interfaces. After all settings are done, configurations should be like shown in the Figure 4, at next page.
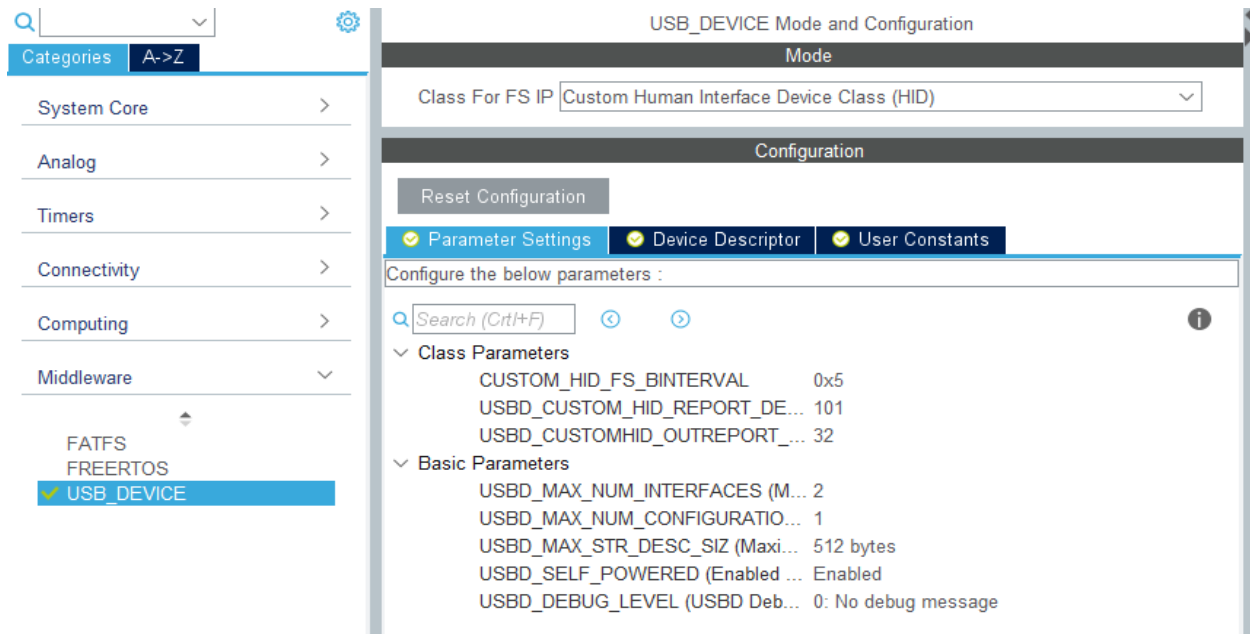
Figure 4 – USB_DEVICE HID Configuration

The "**Device Descriptor**" menu contains the information about your Human Interface Device, namely, VendorID, Language, Manufacturer, ProductID, and other PC side options. You may either change according to your application or leave them. These settings are not in the scope of this project.

UART Communication Configuration

The device should be programmable via UART by using API in order to adjust key functions. Therefore, we need UART interface to be set. Your MCU should support either USART and/or UART. UART configuration is under "**Connectivity**" menu, if you see multiple ports, you may select any not conflicting one. The "**Mode**" parameter of the selected port is set to "**Asynchronous**". If your connection is TTL level, leave RS232 option as "**Disable**". Figure 5 shows the configuration, below.
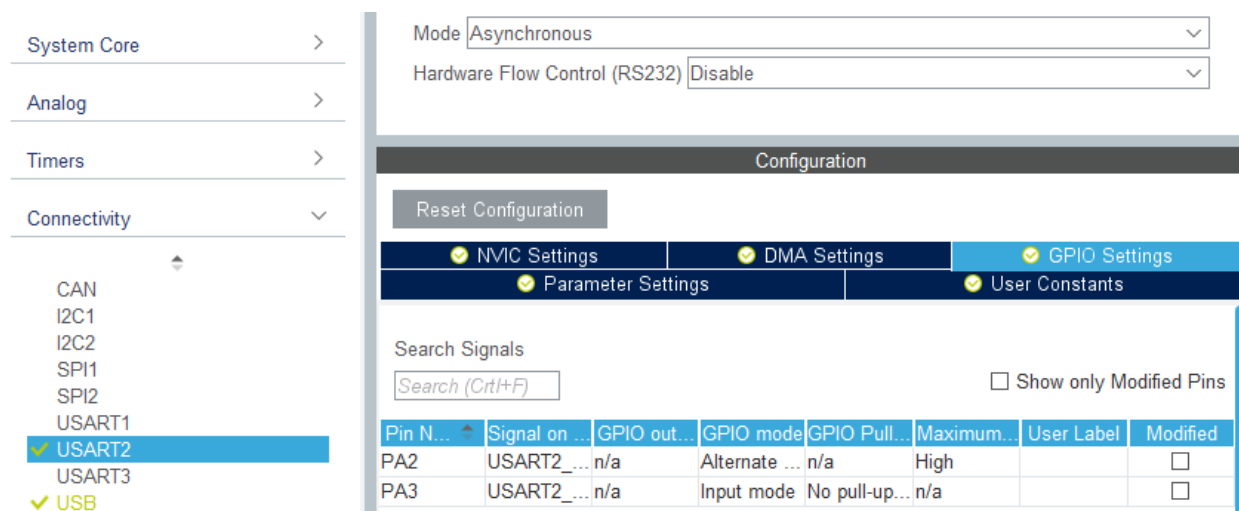
Figure 5 – UART Configuration

The I²C Configuration

This configuration is optional, and to be used for receiving button clicks. Any other method applicable for intended project is sufficient for this purpose. Having 20+ button for this project design, IO Expanders are planned to be used. The selected MCU, - STM32F103C8, supports the High Speed I²C Communication, whose clock frequency is 400KHz.
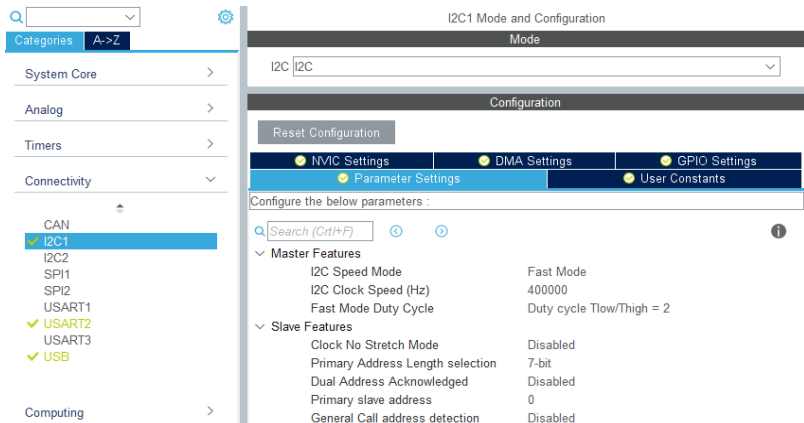
Second Step: Program

When we use "Generate Code" button in STM32CubeMX, we obtain KEIL µVision Project, preset with the MCU configurations we made. In the code side, some parameters should be checked and set properly. The first thing we need to define is the report descriptor, located at "**usbd_custom_hid_if.c**" file. The size of the report descriptor is a macro "**USBD_CUSTOM_HID_REPORT_DESC_SIZE**" defined in "**usbd_conf.h**" file. The size is the one we set in <u>Custom HID Configuration</u> section, and should be 101.

Device Descriptor Configuration

What is Report Descriptor?

The HID interface should be clearly defined in both MCU and CPU, so that proper communication is established. What kind of information, which type of data is sent or received, how many input methods we have, how long a report should be? All are defined in report descriptor. This is like the user guide for PC to understand what MCU is trying to say.

The descriptor definition process is not human readable, and if you see one uncommented, it just looks like a bunch of bytes. The report description is held in an array with size of "**USBD_CUSTOM_HID_REPORT_DESC_SIZE**" parameter, 101 bytes in our case.

Understanding the Structure of Report

We said, we have two interfaces, one acting as mouse, other as keyboard. The mean of this is that we need to define one report description handling both type of reports. Hence, the IN EP (EndPoint) – EP0 is shared with both interfaces for USB Host, but each has different report ID. We can visualize the structure as shown in Figure 6 and Figure 7, below.

Figure 6 – HID Composite Structure

In Figure 7 at next page, we can see an example structure for HID + MSC Composition. We are going to use HID + HID composition.

Figure 7 – HID Composite End Point Structure - Source

What a Pointer give us?

Mouse, is a basic pointer device that we all familiar with. There are two buttons, one pressable wheel, and the position change in 2D. Hence, mouse report consists of 4 bytes containing buttons' states and changes. In Table 1, each information and related bytes / bits are shown. Byte 1-2-3 are signed 8 bit integers, and they represent the change from current position. When using button press actions, be aware of the fact that releasing the button is also require a report as pressing.

Table 1 – Mouse Description Byte Table

Byte	D7	D6	D5	D4	D3	D2	D1	D0
0	----	----	----	----	----	Wheel Button	Right Button	Left Button
1	Signed Relative Change in X							
2	Signed Relative Change in Y							
3	Signed Relative Change in Wheel							

What a Keyboard give us?

Keyboard is the key set we use for typing or for special function keys. We can replicate this functions by sending keyboard reports. Similar to Mouse Buttons, the pressing and releasing the keys are requiring distinct reports. Referring Application Node AN0416 Page 4, the report format is shown in Table 2, below. Since key codes are not equal to ASCII counterpart of keys, you need to use proper HEX Key codes define in USB standards, HID Usage Tables.

Table 2 – Keyboard Description Byte Table

Byte	D7	D6	D5	D4	D3	D2	D1	D0
0	Right GUI	Right ALT	Right SHIFT	Right CTRL	Left GUI	Left ALT	Left SHIFT	Left CTRL
1	RESERVED / PADDING 0x00							
2	KeyCode 1							
3	KeyCode 2							
4	KeyCode 3							
5	KeyCode 4							
6	KeyCode 5							
7	KeyCode 6							

Constructing the Report Descriptor

After we get familiar with the theory behind it, the report descriptor for both mouse and keyboard is configured as shown below:

```
0x05, 0x01,        // Usage Page (Generic Desktop Ctrls)
0x09, 0x06,        // Usage (Keyboard)
0xA1, 0x01,        // Collection (Application)
0x85, 0x01,        //   Report ID (1)
0x05, 0x07,        //   Usage Page (Kbrd/Keypad)
0x19, 0xE0,        //   Usage Minimum (0xE0)
0x29, 0xE7,        //   Usage Maximum (0xE7)
0x15, 0x00,        //   Logical Minimum (0)
0x25, 0x01,        //   Logical Maximum (1)
0x95, 0x08,        //   Report Count (8)
0x75, 0x01,        //   Report Size (1)
0x81, 0x02,        //   Input (Data,Var,Abs,No Wrap,Linear,Preferred State,No
Null Position)
0x95, 0x01,        //   Report Count (1)
0x75, 0x08,        //   Report Size (8)
0x81, 0x01,        //   Input (Const,Array,Abs,No Wrap,Linear,Preferred
State,No Null Position)
0x95, 0x05,        //   Report Count (5)
0x75, 0x08,        //   Report Size (8)
0x15, 0x00,        //   Logical Minimum (0)
0x25, 0x65,        //   Logical Maximum (101)
0x05, 0x07,        //   Usage Page (Kbrd/Keypad)
0x19, 0x00,        //   Usage Minimum (0x00)
0x29, 0x65,        //   Usage Maximum (0x65)
0x81, 0x00,        //   Input (Data,Array,Abs,No Wrap,Linear,Preferred
State,No Null Position)
0xC0,              // End Collection
```

```
    0x05, 0x01,          // Usage Page (Generic Desktop Ctrls)
    0x09, 0x02,          // Usage (Mouse)
    0xA1, 0x01,          // Collection (Application)
    0x85, 0x02,          //   Report ID (2)
    0x09, 0x01,          //   Usage (Pointer)
    0xA1, 0x00,          //   Collection (Physical)
    0x05, 0x09,          //     Usage Page (Button)
    0x19, 0x01,          //     Usage Minimum (0x01)
    0x29, 0x03,          //     Usage Maximum (0x03)
    0x15, 0x00,          //     Logical Minimum (0)
    0x25, 0x01,          //     Logical Maximum (1)
    0x95, 0x03,          //     Report Count (3)
    0x75, 0x01,          //     Report Size (1)
    0x81, 0x02,          //     Input (Data,Var,Abs,No Wrap,Linear,Preferred
State,No Null Position)
    0x95, 0x01,          //     Report Count (1)
    0x75, 0x05,          //     Report Size (5)
    0x81, 0x03,          //     Input (Const,Var,Abs,No Wrap,Linear,Preferred
State,No Null Position)
    0x05, 0x01,          //     Usage Page (Generic Desktop Ctrls)
    0x09, 0x30,          //     Usage (X)
    0x09, 0x31,          //     Usage (Y)
    0x09, 0x38,          //     Usage (Wheel)
    0x15, 0x81,          //     Logical Minimum (-127)
    0x25, 0x7F,          //     Logical Maximum (127)
    0x75, 0x08,          //     Report Size (8)
    0x95, 0x03,          //     Report Count (3)
    0x81, 0x06,          //     Input (Data,Var,Rel,No Wrap,Linear,Preferred
State,No Null Position)
    0xC0,                //   End Collection
    0xC0                 // End Collection
```

If you end up with only uncommented bare HEX form of report descriptors, you may use Report Descriptor Parse to get more human-readable and commented versions.

Great Tool: HID Report Descriptor Decoder

The report descriptor byte array has specific meaning in the USB HID context. Usage Pages, Usages, Inputs, Report Size and Counts are defining the entire package transmission protocol between device and the PC. However, it is hard to obtain a clear structure working with the specific report descriptor by looking the report sizes and counts and their usages. Therefore, for analyzing the descriptors in an easier way, HID Report Descriptor Decoder software is recommended if you are not very familiar with the terminologies. If you want to know more details behind their meaning and constructions, please refer to Related Important Sources section, below.

The mentioned software requires Rexx software to work properly. Please refer to README.md file, located in HIDRDD zip file. After satisfying program requirements, you can see example usages in the same file. In conclusion, you write your report descriptor in a header file (.h) and give this file as an input by using this command:

rexx rd.rex –d –f <input_header_file.h> -o <output_text_file.txt>

The text file gives brief information about the report descriptor and also generates C Structures accordingly, with their meanings.

EndPoint Size Configuration

Since we have both mouse and keyboard report descriptors, as you can see above, **Report ID** is added in order to distinguish the inputs between mouse and keyboard. Of course, this **Report ID** byte is also added to the report size, expectedly. Eventually, while mouse report has 5 bytes, keyboard report has 8 bytes, without one **KeyCode** byte. Therefore, we need to adjust "**USB_MAX_EP0_SIZE**" parameter, located at "**usbd_desc.c**" file, to the value **8U**, meaning each packet only can have maximum 8 bytes.

Related Important Sources

1) STM32 USB-FS-Device - UM0424 - Link
2) USB HID demonstrator - UM0551 - Link
3) HUMAN INTERFACE DEVICE TUTORIAL - AN249 - Link
4) STM32F103x8 - STM32F103xB –Datasheet - Link
5) USB – HID USAGE TABLES - Link

Third Step: Sending Reports

After all the configurations, we are ready to publish instructive reports to activate specified key event on the CPU side. Key events are defined namely, "**KeyDown**", "**KeyPress**", and "**KeyUp**", if you are not familiar with these terminologies and their functionalities, please refer to Microsoft Documentations.

Report Send Command

Since we have only one IN EndPoint shared both Keyboard and Mouse Reports, one function handles both communications. The "**USBD_CUSTOM_HID_SendReport**" function located at "**usbd_custom_hid_if.c**" file, takes "**USBD_HandleTypeDef**" type object that we obtain from USB HID initiation, in "**usbd_device.c**" file. The other two input parameters are related to the report we send. First, we pass the report structure we set and then the length of the report.

Mouse Reports

The mouse report structure is elaborated in the Pointer Section, in the case of ambiguity please refer above. Hence, both the mouse and the keyboard report are handled in one EP, ReportID byte is added to the structure to distinguish between them. The other parameters are the same as discussed. Be cautious and aware of signed "**dx**", "**dy**", and "**dWheel**" parameters while using.

Keyboard Reports

The content of the keyboard reports is discussed in the Keyboard Section, please refer if you have difficulties. The reserved byte should be 0x00 always, this functions as padding between modifier byte and KeyCodes. Please give attention to the fact that the release action is also requeires report as pressing action, while using the keyboard reports.

Report Data Types

Mouse Report Structure

Each report has its unique Report ID byte to distinguish between other report types. Every detail of this structure is already covered in Mouse Reports section.

```c
typedef struct{
    uint8_t   reportID;
    uint8_t   buttonMask;
    int8_t    dx;
    int8_t    dy;
    int8_t    dWheel;
} MouseReport_t;
```

Keyboard Report Structure

Each report has its unique Report ID byte to distinguish between other report types. Every detail of this structure is already covered in Keyboard Reports section. This structure has been change to be scaled to five bytes by reducing unused three **KeyCode** byte. In result, we have mouse and keyboard report both having five-byte-structure.

```c
typedef struct{
    uint8_t   reportID;
    uint8_t   modifier;
    uint8_t   _RESERVED;
    uint8_t   byte1;
    uint8_t   byte2;
    //uint8_t  backupKeys[3]; // In case of need, Plese modify the
ReportDescriptor & Max_EPIN_Size accordingly.
} KeyboardReport_t;
```

When to send reports?

The importance of the capability to send or in other words successively get reaction from the CPU is related to the "**EP buffer**", "**bInterval**", and the "**PoolingInterval**" of the HID configurations. For further details please refer to AN249, in the Related Important Sources Section. The sources given are highly recommended to investigate for all parts discussed so far.

If the USB HID Peripheral is busy, the "**USBD_CUSTOM_HID_SendReport**" function may return "**USBD_BUSY**" value, and should be handled manually. In need of custom SendReport function, you may find "**USBD_CUSTOM_HID_SendReport_FS**" function in the "**usbd_custom_hid_if.c**" file.

An Example of Sending Report

```c
USBD_CUSTOM_HID_SendReport(&hUsbDeviceFS,    (uint8_t   *)    &mouse_report,
sizeof(MouseReport_t));
```

API Structure

Why do we need an API?

The second phase of the project should satisfy the "programmable keys" requirement. The number of keys to be used is not specified and assumed to be more than 20. Each key has an action by default and can be assigned another action later by using this API, via UART.

Environment of the API

Because of the high key number, this project is shaped upon I^2C Communication. The ongoing stages of the project assumes that key press actions to MCU is obtained from I^2C bus – an example: ExpenderIO. The GUI is designed for Windows, by using C# and Visual Studio IDE. For testing purposes, the GUI has only eight button.

Main Windows

The Main Window contains the testing buttons; each has an action assigned by default. The default settings and button action memory is kept in "**Keys.xlsx**" Excel file, which will be elaborated later. Main page allow the user select the button which is going to be configured. In the case of one key is pressed "**Button Setting Window**" appears.

Figure 8 – KeySim API – Main Window

Button Setting Window

When one of the buttons are clicked, meaning that the user wants to re-configure this buttons action, "**Button Setting Window**" is shown. The "**Name**" parameter is descriptive information about the key to remind the user about this function.

The "Function" parameter is a drop-list containing three elements, namely, "**Keyboard**", "**Mouse**", and "**Custom**". This parameter specifies the action domain of the button, is of course related to ReportID for MCU. In the case of "**Keyboard**" or "**Mouse**" is selected, the Value drop-list is re-constructed upon the action domain. In other words, Value drop-list only contains the actions belonging to function selected.
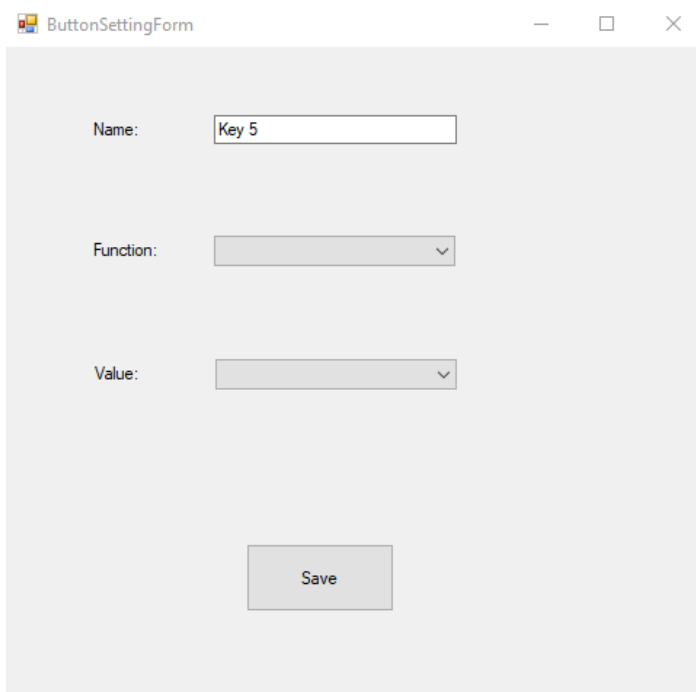

Figure 9 - The Button Setting Window

API Data Structure

One of the important information to store is the key functions. To store key specifications, Excel format is selected.

Keys Excel File

For a key, it is sufficient to store the **ReportID** (Function), Report (Value), and ID (To localize and match the key in MCU side).

Both Mouse and Keyboard reports are discussed elaborately in previous sections. However, for mouse since we only interested in click actions, only one byte is sufficient. In the keyboard case, one modifier byte and one **KeyCode** byte is

ID	Name	ReportID	Byte1	Byte2
0	Key 1	1	0	4
1	Key 2	1	2	4
2	Key 3	2	1	0
3	Key 4	2	2	0
4	Key 5	1	0	8
5	Key 6	1	0	9
6	Key 7	1	0	10
7	Key 8	1	0	11

sufficient to perform all necessary actions. These default settings are configured based on first phase of the project. If **ReportID** and Bytes are causes ambiguity, please refer to Mouse Reports and Keyboard Reports, and Related Important Sources sections.

KeyClass Class

The **KeyClass** class is the model for each key represented in the project. The KeyClass class is compatible with the Excel file described above. This class is the bridge we use between API and Excel.

```
class KeyClass
    {
     public Byte UniqueID { get; set; }    // ArrayID for MCU
     public string Name { get; set; }
     public Byte ReportID { get; set; }    // Keyboard (0x01)  | Mouse (0x02)
     public Byte Byte1 { get; set; }       // Modifier Byte    | Button Byte
     public Byte Byte2 { get; set; }       // KeyCode 1        | NONE
    }
```

The **ReportID** contains the information that "**Keyboard**" or "**Mouse**" is chosen. Byte1 represents the "**Modifier**" Byte for Keyboard Report, whereas "Button" Byte for mouse Report. Byte2 is only applicable for Keyboard Reports holding the **KeyCode**, and meaningless for Mouse Reports.

ComboBoxItem Class

The "**Value drop-list**" is **ComboBox** type list. The **ComboBoxItem** class is constructed to achieve robust and intuitive understanding for action options. The "**Text**" parameter of the class is the option name which the user see in the "**Value drop-list**". The "**Value**" parameter of the class is holding the Report Details for selected action.

```
public class ComboBoxItem
{
    public string Text { get; set; }
    public struct ValueStructure
    {
        public ValueStructure(Byte reportID, Byte byte1, Byte byte2)
        {
            ReportID    = reportID;
            Byte1       = byte1;
            Byte2       = byte2;
        }

        public Byte ReportID { get; set; }
        public Byte Byte1 { get; set; }
        public Byte Byte2 { get; set; }

    }

    public ValueStructure Value { get; set; }

    public override string ToString()
    {
        return Text;
    }

    public ComboBoxItem(string text, ValueStructure value)
    {
        this.Text = text;
        this.Value = value;
    }
}
```

Keys List and Constant List

This API contains two type of list, namely, Keys list, and constant lists. The **KeyClass** is said to be bridge between Excel File and API. This is because, the Keys list is made of **KeyClass** Objects derived from the Excel File discussed in the Data Structures section above.

The constant list is used to limit user interactions and modifications in order to prevent the program from misuse or abuse actions. The constant lists are "**Keyboard Actions List**" and "**Mouse Actions List**", predefined and static for this project scope.

In addition, one helper function is needed to evaluate and prepare the excel file to be useable for reading and formatting lists. The helper function should construct **DataTable** type from the Excel File.

Helper Function: ReadExcelFile

```csharp
private static DataTable ReadExcelFile(string sheetName, string path)
{
    using (OleDbConnection conn = new OleDbConnection())
    {
        DataTable dt = new DataTable();
        string Import_FileName = path;
        string fileExtension = Path.GetExtension(Import_FileName);
        if (fileExtension == ".xls")
            conn.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" + Import_FileName + ";" + "Extended Properties='Excel 8.0;HDR=YES;'";
        if (fileExtension == ".xlsx")
            conn.ConnectionString = "Provider=Microsoft.ACE.OLEDB.12.0;Data Source=" + Import_FileName + ";" + "Extended Properties='Excel 12.0 Xml;HDR=YES;'";
        using (OleDbCommand comm = new OleDbCommand())
        {
            comm.CommandText = "Select * from [" + sheetName + "$]";
            comm.Connection = conn;
            using (OleDbDataAdapter da = new OleDbDataAdapter())
            {
                da.SelectCommand = comm;
                da.Fill(dt);
                return dt;
            }
        }
    }
}
```

This function is general purpose helper function to build **DataTable** type variable. The source of the function is a StackOverflow Mention.

KeyList List

The **KeyList** list is containing all the buttons defined in [Excel File](#). This list is used for reading proper and intuitive data and re-configuring stages. The construction process of the KeyList list is shown below:

```csharp
private   static   readonly   DataTable   dataTable1=   ReadExcelFile("Sheet1",
"C:/Users/ataberk.oklu/Documents/Keys.xlsx");

public object[] ValueComboBoxList = new object[] { };

List<KeyClass> KeyList = dataTable1.AsEnumerable().Select(row =>
   new KeyClass {
       UniqueID = Convert.ToByte(row[0]),
       Name     = Convert.ToString(row[1]),
       ReportID = Convert.ToByte(row[2]),
       Byte1    = Convert.ToByte(row[3]),
       Byte2    = Convert.ToByte(row[4])
   }).ToList();
```

Each row is enumerated and constructed and KeyClass object for each row from the Excel File.

Keyboard Actions List

This action list only contain the action are possible with keyboard and restricted by the project contains. In other words, the user cannot select an action other than predetermined by project. This List can be modified according to project requirements. However, the manipulator should obey report descriptor structure and byte meanings strictly, to avoid malfunctioned results.

```csharp
private static readonly ComboBoxItem[] keyboardActionsList = new ComboBoxItem[]
{
  new ComboBoxItem("A", new ComboBoxItem.ValueStructure(0x01, 0x02, 0x04)),
  new ComboBoxItem("B", new ComboBoxItem.ValueStructure(0x01, 0x02, 0x05)),
  new ComboBoxItem("C", new ComboBoxItem.ValueStructure(0x01, 0x02, 0x06)),

  new ComboBoxItem("a", new ComboBoxItem.ValueStructure(0x01, 0x00, 0x04)),
  new ComboBoxItem("b", new ComboBoxItem.ValueStructure(0x01, 0x00, 0x05)),
  new ComboBoxItem("c", new ComboBoxItem.ValueStructure(0x01, 0x00, 0x06)),

  new ComboBoxItem("DEL", new ComboBoxItem.ValueStructure(0x01, 0x00, 0x4C)),
  new ComboBoxItem("BCS", new ComboBoxItem.ValueStructure(0x01, 0x00, 0x2A)),
  new ComboBoxItem("SPC", new ComboBoxItem.ValueStructure(0x01, 0x00, 0x2C))
};
```

Only nine actions are specified for testing purposes. Again, in the case of manipulation and/or addition, please refer to related sections and [USB HID Usage Tables](#) in the [Related Important Sources](#) section.

Mouse Action List

In the scope of this project, only the "**Right Click**" and" **Left Click**" actions are implemented for mouse:

```
private static readonly ComboBoxItem[] mouseActionsList = new ComboBoxItem[] {
    new ComboBoxItem("Right Click", new ComboBoxItem.ValueStructure(0x02, 0x01,
0x00) ),
    new ComboBoxItem("Left Click", new ComboBoxItem.ValueStructure(0x02, 0x02,
0x00) )
};
```

API Pipeline

First Step: Button Selection

The user selects the button to re-configure via Main Window interface. Each Button is configured symmetrically to each other and have the same pattern. In the case of addition, a new button, the only configuration is adding click control and showing the "**ButtonSettingForm**". Each Button pass their "**UniqueID**" or "**ArrayID**" to the "**ButtonSettingForm**", in order to obtain which button is selected.

Second Step: Button Configuration

In the "**Button Setting Window**", "**Name**" textbox is filled with the last configuration. This is descriptive information for the user. The "**Function ComboBox**" is selected according to the last assigned function for the button, with the help of "**ReportID**" parameter of the selected button. Lastly, the "**Value**" parameter, keeping the action or report information for the button. The constant action lists are listed according to "**Function ComboBox**" selection.

After the re-configuration is done, the user can save and re-program the MCU by clicking "**Save**" button, at the bottom of the form.

Third Step: Saving and Sending New Button Configurations

The last configuration of the only changed button is written to the **Excel File** on the related row, and saved. After that each button objects' "**UniqueID**", "**ReportID**" and "**Bytes**" are serially transmitted to the MCU, via **UART**. **UniqueID** parameter is being used to locate and synchronize the buttons, which are locally stored in the **Flash Memory** of the MCU.

Saving to Excel

The first phase of keeping the information of the modified key is saving to the Excel File. Since the selected button data is passed to the "**Button Setting Window**" from the "**Main Window**", we can use its "**UniqueID**" to find where the associated **KeyClass** object is located in which row in the Excel File.

It is good to be aware of the indexing format of the Excel Cells. The first element, - the most upper-right, is indexed as **[1, 1]**, indicating 1st Row and 1st Column of the table. Therefore, the excel location of any **KeyClass** Object is **two** row shifted from the beginning, one for the header row, and the other is for the indexing format of the Excel.

```csharp
private void Save_button_Click(object sender, EventArgs e)
{
    Excel.Application xlApp = new Excel.Application();
    Excel.Workbook                          xlWorkBook                          =
xlApp.Workbooks.Open("C:/Users/ataberk.oklu/Documents/Keys.xlsx");
    Excel.Worksheet xlWorkSheet = xlWorkBook.Worksheets["Sheet1"];
    int SelectedRow = KeyList[buttonNumber].UniqueID + 2;
    ComboBoxItem SelectedKey = ValueComboBox.SelectedItem as ComboBoxItem;

    xlWorkSheet.Cells[SelectedRow, 2].value = name_textbox.Text;
    xlWorkSheet.Cells[SelectedRow, 3].value = SelectedKey.Value.ReportID;
    xlWorkSheet.Cells[SelectedRow, 4].value = SelectedKey.Value.Byte1;
    xlWorkSheet.Cells[SelectedRow, 5].value = SelectedKey.Value.Byte2;

    xlWorkBook.Close();
    xlApp.Quit();

    System.Runtime.InteropServices.Marshal.ReleaseComObject(xlApp);
    System.Runtime.InteropServices.Marshal.ReleaseComObject(xlWorkBook);
    System.Runtime.InteropServices.Marshal.ReleaseComObject(xlWorkSheet);
}
```

Sending to the MCU via UART

The UART Communication for this project and the selected MCU is defined as TTL Level. The UART Communication is configured in the MCU Configuration section. The API send each **KeyClass** objects with their **UniqueID**, **ReportID**, and two **ReportBytes** in four-byte-frame, called "**Key_UART_Frame**". Each received frame is handled in MCU side, to store local flash memory to accessed each startup regime.

The main structure of the "**Key_UART_Frame**" is as shown below:

```csharp
ComboBoxItem SelectedKey = ValueComboBox.SelectedItem as ComboBoxItem;

Byte[] Key_UART_Frame = {
    KeyList[buttonNumber].UniqueID,
    SelectedKey.Value.ReportID,
    SelectedKey.Value.Byte1,
    SelectedKey.Value.Byte2
};
```

Every "Save" action triggers the MCU to receive one "**Key_UART_Frame**":

```
SerialPort _serialPort        = new SerialPort();
_serialPort.PortName          = "COM2";
_serialPort.BaudRate          = 115200;
_serialPort.Parity            = Parity.Even;
_serialPort.DataBits          = 8;
_serialPort.StopBits          = StopBits.One;
_serialPort.Handshake         = Handshake.None;
_serialPort.WriteTimeout      = 500;

try
{
    if (!(_serialPort.IsOpen))
        _serialPort.Open();
    _serialPort.Write(Key_UART_Frame, 0, 4);
}
catch (Exception ex)
{
    MessageBox.Show("Error opening/writing to serial port: " + ex.Message,
"Error!");
}
```

The "**Communication Port**" setting will be manually and static defined according to project specifications. The "**Bound Rate**", "**Parity**", "**DataBits**", and "**Stop-bit**" parameters are set according to UART Configuration Section.

Forth Step: Receiving and Writing to the Flash

The maintenance UART port is planned to be used to update key configurations located in the Flash Memory of the MCU.

UART Reception

Each change made by using API results in sending only changed key's configuration, which consists of four bytes describing which key it is and necessary report information. Since this sent "**KeyFrame**" is discussed API – Sending section, please refer for detailed descriptions.

The reception of these four bytes can be handled by three different methods, namely, "**blocking receive**", "**Non-blocking with Interrupt**", and "**DMA Reception**". Blocking Reception method bases on the given timeout and the number of received bytes. The MCU will not do any work until either it receives given amount of bytes or the given timeout is reached. The reception with interrupt does not block the main processes of the MCU. When new byte is received, the interrupt occurs and handles the received byte, then returns to the last main program vector with the help of NVIC. The use of "**Direct Memory Access**" method allow the reception has no effect on the main program flow. The DMA has the ability to handle both "**peripheral – memory**" and "**memory – memory**" writing and reading requests.

In this project DMA method is used, just in order to practice DMA Structure. While the MCU is re-programming, by assumption, even blocking reception method can be used, since no active use of the keys is not expected. When four expected bytes are received, DMA triggers the "**RX Transfer Completed Callback**", - **HAL_UART_RxCpltCallback** function defined as __weak. Since this callback function is defined as weak, we can overwrite this function in the **main.c** file in order to handle the received "**KeyFrame**".

First of all, the DMA RX process should be activated in the "**Initiation Portion**" in the main function, in other words, in **USER CODE 2** Section.

```
HAL_UART_Receive_DMA(&huart2, UART2_rxBuffer, NUM_OF_BYTES_IN_KEY_FRAME);
```

Then, we need to overwrite RX Callback Function to set our "**KeyList**" with the change.

```
typedef union {
    KeyboardReport_t keyboardReport;    // 5 Bytes
    MouseReport_t mouseReport;          // 5 Bytes
} ReportUnion;
ReportUnion KeyList[NUMBER_OF_KEYS];


void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
        if(UART2_rxBuffer[1] == 0x01){          // Unique ID UART2_rxBuffer[0]

            KeyList[UART2_rxBuffer[0]].keyboardReport.reportID  = UART2_rxBuffer[1];
            KeyList[UART2_rxBuffer[0]].keyboardReport.modifier  = UART2_rxBuffer[2];
            KeyList[UART2_rxBuffer[0]].keyboardReport.byte1     = UART2_rxBuffer[3];

        } else if(UART2_rxBuffer[1] == 0x02){

            KeyList[UART2_rxBuffer[0]].mouseReport.reportID     = UART2_rxBuffer[1];
            KeyList[UART2_rxBuffer[0]].mouseReport.buttonMask   = UART2_rxBuffer[2];

        }
    HAL_UART_Receive_DMA(&huart2, UART2_rxBuffer, NUM_OF_BYTES_IN_KEY_FRAME);
}
```

The "**ReportUnion**" union is used to increase readability of the code and make the code more organized. "**KeyList**" array is constructed in the same order as The Excel File, discussed in API Section. Array index refers to the Key ID, which is send as first byte in "**KeyFrame**". The Report ID follows, and finally two report bytes are placed.

Writing to the Flash Memory

Writing process to the Flash Memory requires more care and extra work to do compared to the SRAM counterpart. This report assumes the reader has sufficient knowledge about flash memory structure, addresses, pages / sections of the target MCU. Since the flash memory knowledge is not in the scope of this report, please refer to the datasheet of the MCU. On the other, the basic procedure of the writing process to the Flash Memory can be explained as follows:

- Unlock the Flash
- Erase the page or section to be used to store KeyList
- Write the KeyList array with care of the capabilities and restrictions of the target MCU
- Lock the Flash

For each action you can use either HAL Functions or handle by manually under the guide of the datasheet. The Unlocking and Locking Keys should be represented in the datasheet.

```
#define FLASH_UNLOCK_KEY1   0x45670123
#define FLASH_UNLOCK_KEY2   0xCDEF89AB
```
For STM32F103C8 MCU.

It is minor but important detail to mention why we write the "**KeyList**" into Flash Memory. The SRAM is dedicated to program variables used in the program, and when the device is reset, all data kept in SRAM lost. The data in flash memory is kept until the data is erased by using the same method above or other debug tools.