NORDIC SEMICONDUCTOR

Generic data transfer using USB HID

nAN-22

Application Note v1.0

Keywords

- Generic data transfers to and from nRF24LU1+
- Simpler alternative for applications with small bandwidth requirements
- Built-in HID drivers in Microsoft Windows®/OS X®/Linux
- Fully documented Microsoft Windows example program with source code
- Free development platform, Visual C# express from Microsoft

All rights reserved
Reproduction in whole or in part is prohibited without the prior written permission of the copyright holder.

2011-09-06

Contents

1 Introduction 3
2 USB-HID 4
2.1 When to use HID 4
2.2 HID descriptors 4
3 Project setup..6
3.1 Required hardware 6
3.2 Required software..6
3.3 Project structure...6
4 Implementation...7
4.1 nRF24LU1+ USB HID firmware 7
4.1.1 Configuration 7
4.1.2 USB descriptors 8
4.1.3 HID descriptor 9
4.1.4 Report descriptor (g_usb_hid_report_data) 10
4.2 nRF24LU1+ application specific firmware 11
4.3 Host application: interaction with the USB HID library 11
4.3.1 Configuration 11
4.3.2 Sending data to nRF24LU1+...13
4.3.3 Receiving data from nRF24LU1+ 13
4.4 Host application: overview 13
4.4.1 Main form 14
4.4.2 Progress window 15
4.5 Host application structure 15
4.5.1 Communication 16
4.5.2 Loading a HEX file...16
4.5.3 Transferring the loaded firmware 16
4.5.4 Verifying the remote firmware 16
5 References 16

1 Introduction

The Universal Serial Bus is the de facto industry standard for the connection of computer peripherals, with approximately two billion sold devices every year. The success of USB is in large part due to its end product ease of use. However, such simplicity comes at a price, as the developer is often burdened with creating complex USB drivers. Fortunately, there is a simpler way to provide devices with USB support.

To avoid the complexity of creating USB drivers when working with generic data transfers, you can use the standard HID class of USB devices, with built-in drivers in all modern operating systems.

This application note primarily describes how to use HID to transfer data between a Microsoft Windows application and the nRF24LU1+. It also goes into some detail about the example application, an over-the-air firmware flash updater for the nRF24LE1 using the combined USB and RF features of the nRF24LU1+. The specifics of the radio transmission protocol and the nRF24LE1 firmware are discussed in another application note, nAN-18.

The implementation sections for the nRF24LU1+ and the host application are separated into two subsections: One subsection describing details of using the HID class for USB data transfers, and one explaining the specifics of the firmware update application. These sections correspond to the marked parts of Figure 1.

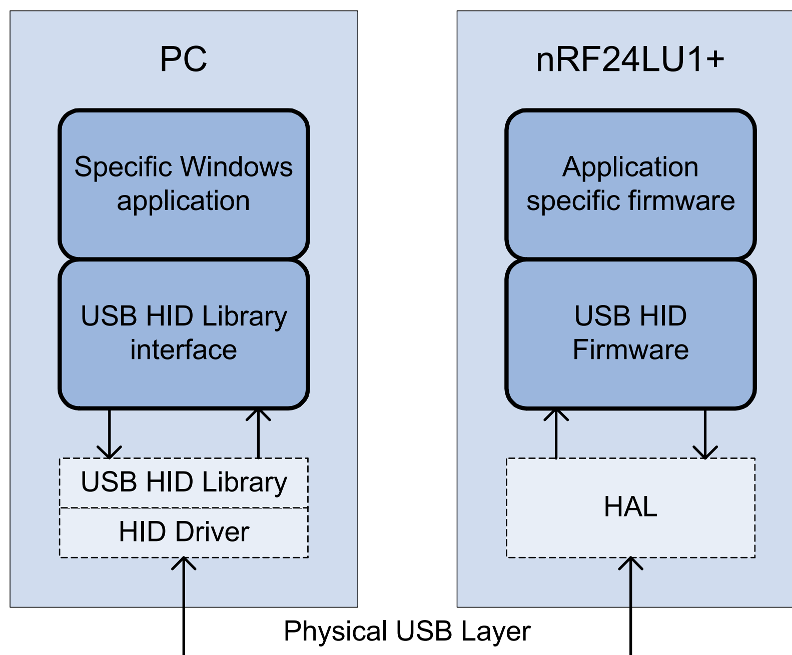

Figure 1. Implementation split into PC part and nRF24LU1+ part

This application note assumes that the reader is familiar with the following topics:

- Windows Forms development using C#
- C programming of Nordic Semiconductor devices using Keil™µVision
- USB protocol and specification, including basic understanding of central subtopics like enumeration, transfer types, endpoints, and descriptors

Terminology from these technologies is used throughout this document. Users unfamiliar with this terminology can refer to documentation listed in the references section of this application note.

2 USB-HID

HID is a standard USB device classification meant to include all kinds of Human Interface Devices, such as computer keyboards and mice, medical instruments and video game controllers. The HID class provides great flexibility by incorporating the concept of Reports containing the transferred data.

HID devices provide descriptors for these reports to the host, which contain information about what kind of data they represent, such as mouse cursor movement. By describing the reports as being vendor specific, it is possible to use them as simple data packets, leaving the interpretation of the contents completely up to the specific application.

HID only supports Interrupt transfers in addition to the obligatory control transfers. This implies guaranteed transfer intervals, but also limited bandwidth as the minimum transfer interval is 1 ms, and the packet size is restricted to a maximum of 64 bytes.

2.1 When to use HID

There are several advantages, but also some disadvantages to using HID for your generic data transfers. In simple terms, if you can live with the bandwidth restrictions, you should seriously consider using HID, unless you have the time and desire to write your own custom drivers.

Here is a short list of pros and cons you should be aware of:

Pros:

- Existing drivers drastically reduce development effort.
- Guaranteed data transfer rate and latency with USB Interrupt transfers.
- Platform independent standard, supported by all modern operating systems.

Cons:

- Maximum theoretical data transfer rate of 64 kB/s.

2.2 HID descriptors

All USB devices must have a set of descriptors defining various parameters and information about them. At enumeration, which happens upon connection of the device, these descriptors are requested by the host. The host uses this information to determine the drivers needed to interact with the device, which in turn might request even more specific descriptors.

The obligatory descriptors that must be provided by all USB devices are displayed on the left side of Figure 2. on page 5, while the descriptors specific to HID standard devices are displayed on the right, inside the dotted line. If you are unfamiliar with the standard USB descriptors, we encourage you to read up on them in one of the relevant links in the references section.

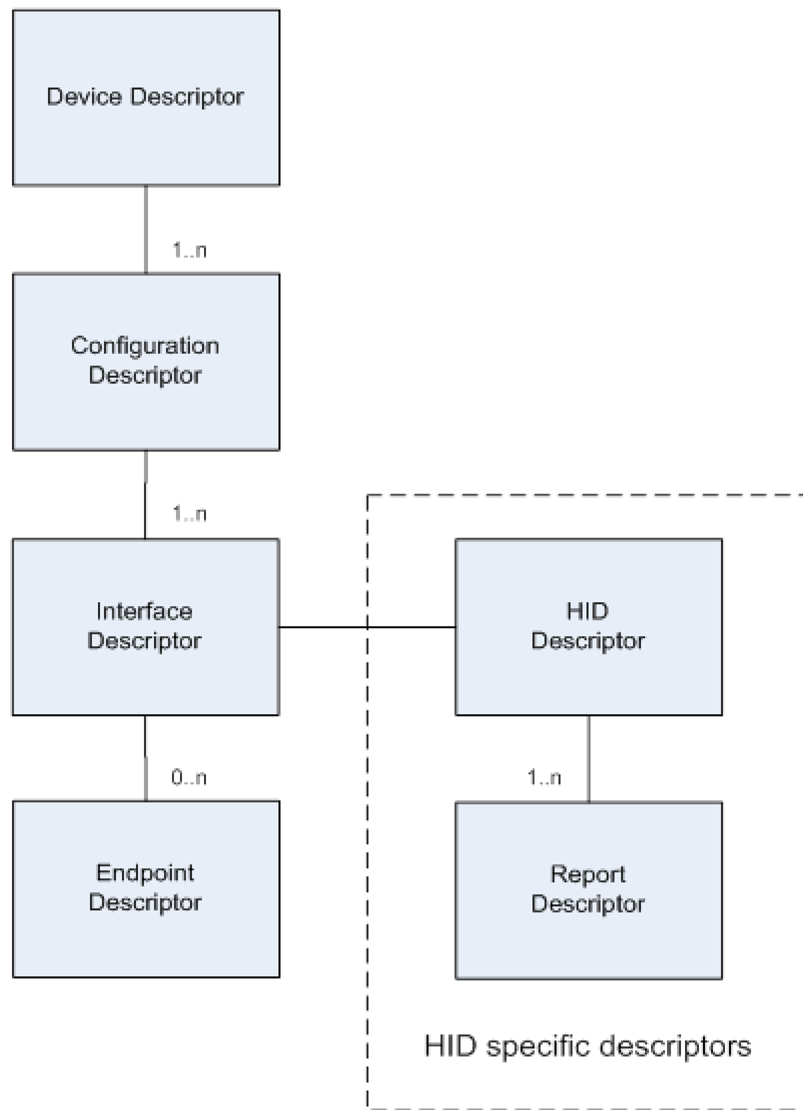

Figure 2. USB descriptors

As a part of the interface descriptor it is possible to specify a standard USB class, such as HID, which usually also determines what drivers the host should load. In the case of a HID device, the built-in drivers of the operating system will request a specific set of HID descriptors. This includes a HID descriptor, at least one report descriptor, and lastly, an optional set of physical descriptors which are not considered in this application note.

The HID descriptor essentially works as a table of contents for the report and physical descriptors by specifying their count, types, and sizes. Our application-specific HID descriptor is described with an explanation of the fields in the implementation section.

Report descriptors, as mentioned earlier, describe the format and the meaning of data sent and received by the HID class device. A report descriptor consists of several fields called items which are used together with corresponding data fields to describe various aspects of what the report data represents. The number of items included in a report descriptor is variable, making the amount of details highly flexible. The example report descriptor used by our application is given in the implementation section, along with brief

explanations of fields. The interested reader will find more information on report descriptors in the HID specification.

3 Project setup

This application note and the attached code are part of a project which describes how to update application-specific firmware over RF. To fully utilize the project to update application-specific firmware over RF, you will need the hardware, software and project structure listed in this chapter. See the application note nAN-18 for further reference.

To set up the attached project correctly you will need the following hardware, software, and project structure. If you are only interested in the USB HID part, there is no need for the nRF24LE1-specific hardware which is referred to within parantheses in the following lists.

3.1 Required hardware

- nRFgo Development Kit nRF24LU1P-FxxQ32-DK, nRF24LU1+
- (nRFgo Development Kit nRF24LE1-F16Qxx-DK, nRF24LE1)
- nRF6310 Motherboard (x2)
- PC workstation with USB

3.2 Required software

- Keil µVision V4
- C51 Compiler
- BL51 Linker
- nRFgo Software Development Kit, version 2.2
- Windows 7: 32-bit, 64-bit
- Microsoft Visual C# 2010 Express

3.3 Project structure

Firmware updater

- (Bootloader nRF24LE1)
- Common
- USB-RF adapter nRF24LU1+
- Host application
- Precompiled HEX

If you have not already done so, you should install Keil µVision and the nRFgo Software Development Kit (SDK). Once you have done this, you can place the Firmware_updater folder in the …\nRFgo SDK 2.2.0.270\source_code\projects\nrfgo_sdk folder. This will ensure that the predefined project files will have the correct include paths to compiler- and hal-directives. If you wish to place the project at a different location you will have to set the include paths manually in Keil µVision. These are found under Project – Options for Target '…' - C51 – Include Paths.

You will find the Keil µVision project files for the nRF24LE1 update firmware in the boot loader nRF24LE1 folder. A precompiled HEX file for the nRF24LE1 update firmware is found in the precompiled HEX folder. This HEX file can be flashed directly to the chip if you do not want to build the project files before you test out the functionality.

Demo firmware that can be used as the new firmware, is found in the Precompiled-HEX folder.

4 Implementation

4.1 nRF24LU1+ USB HID firmware

Focusing on the USB handling, the nRF24LU1+ firmware main loop can be visualized like the flowchart of Figure 3. with a USB routine manipulating the application state based on the received commands and data. This basic program flow is very simple, and you can use it for your own applications using HID. The setup of the USB HID communication is, unfortunately, not so simple. Using Hardware Abstraction Layer (HAL) functions simplifies setup somewhat, but one still has to tailor the USB descriptors to make the application function and appear correct to the host.

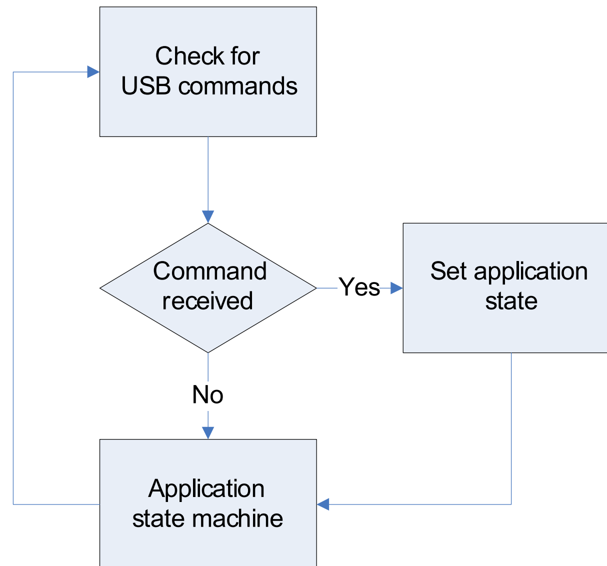

Figure 3. USB firmware flowchart

4.1.1 Configuration

To initiate the USB controller, we call the function hal_usb_init in our main function, which registers user-defined callback functions:

```
// USB HAL initialization
hal_usb_init(true, device_req_cb, reset_cb, resume_cb, suspend_cb);
```

The important thing to note here when using HID is the function device_req_cb which must handle USB requests specific to the HID subsystem. Our device_req_cb function simply forwards the request to the HAL function hal_usb_hid_device_req_proc which takes care of this for us. The three other callbacks let you implement functionality on the reset, resume, or suspend USB commands. The init function also loads the descriptors which are discussed in the following subsection.

Next, we configure the endpoints, in our example one IN endpoint, and one OUT endpoint:

```
hal_usb_endpoint_config(0x81, EP1_2_PACKET_SIZE, ep_1_in_cb);    hal_usb_endpoint_config(0x02,
EP1_2_PACKET_SIZE, ep_2_out_cb);
```

These functions set the endpoint addresses, maximum sizes, and callbacks which will be called when a transaction in either direction completes. For OUT endpoint callbacks (directed to our device), it is a good practice to copy the data from the USB buffer and set a received data flag. This way, the USB buffer is quickly made ready for another packet. For IN endpoint callbacks, you should set a flag indicating that data

has been sent. By checking this flag before you try to send more data, you avoid undefined behavior when trying to send when a transaction is already in progress.

4.1.2 USB descriptors

usb_desc.c contains all the descriptor definitions for our device, while some descriptor structures and central parameters are defined in usb_desc.h for easier modification. If you are looking to make some small changes, you can modify the packet size, polling interval, Vendor ID and Product ID, just by editing the constants in the header file.

For more significant changes like adding interfaces or endpoints, you will need to modify the descriptor structures directly. The following tables give brief explanations of the essential descriptor structures found in the two descriptor files:

4.1.2.1 usb_desc.h

Structure	Description
usb_conf_desc_templ_t	Declaration of the configuration descriptor structure. This must declare fields for the basic configuration descriptor, all interface descriptors, HID descriptors, and endpoint descriptors. You have to modify this if you want to change the number of interfaces or endpoints.
VID	Vendor ID Constant used in the device descriptor.
PID	Product ID Constant used in the device descriptor.
MAX_PACKET_SIZE_EP0	Constant defining endpoint 0 maximum packet size. This must be at least the size of the largest other endpoint.
EP1_2_PACKET_SIZE	Constant defining endpoint 1 and 2 maximum packet size, as well as the HID report sizes. For our implementation this can be maximum 32 bytes.
EP1_POLLING_INTERVAL	Constant defining milliseconds between interrupt messages for endpoint 1.
EP2_POLLING_INTERVAL	Constant defining milliseconds between interrupt messages for endpoint 2.
USB_STRING_DESC_COUNT	Constant defining the number of used string descriptors.

Table 1. Descriptor structures of usb_desc.h

4.1.2.2 usb_desc.c

Structure	Description
g_usb_dev_desc	Specification of the device descriptor
g_usb_conf_desc	Specification of the configuration descriptor. Include specifications of the interface, HID, and endpoint descriptors.
g_usb_hid_report_data	HID report descriptor for our vendor-specific data endpoints. If you want to specify your own report descriptors, you must make similar structures.
g_usb_hid_hids	Contains all HID descriptors, along with their report descriptors.
g_usb_string_desc	Array of the string descriptors. Observe the format of the string descriptor and remember that the size must include the descriptor type and size itself.

Table 2. Descriptor structures of usb_desc.c

4.1.3 HID descriptor

Our HID descriptor is shown in Table 3.. It is fairly straightforward, and all you have to do to add more report (or physical) descriptors, is to increase the bNumDecriptors field, and add the type and length fields for all of them. This descriptor is defined as a part of the configuration descriptor (g_usb_conf_desc) found in usb_desc.c.

Field	Value	Description
bLength	sizeof(hal_usb_hid_desc_t)	Size in bytes of this descriptor
bDescriptorType	USB_CLASS_DESCRIPTOR_HID	HID descriptor identifier. (0x21)
bcdHID	0x0110	Specify HID version 1.11
bCountryCode	0x00	Set country code to 0, as it is not relevant for our device.
bNumDescriptors	0x01	The number of following HID specific descriptors
bDescriptorType	USB_CLASS_DESCRIPTOR_REPORT	The first following HID specific descriptor is a report descriptor. (0x22)
wDescriptorLength	sizeof(g_usb_hid_report_data)	Size of our report descriptor

Table 3. HID descriptor

4.1.4 Report descriptor (g_usb_hid_report_data)

Even though we are only considering simple data transfers in this application note, we still need a report descriptor containing the bare minimum required for it to be recognized as a HID device. We have included it in section 4.2 on page 11, with a brief explanation of its fields.

The report descriptor items start with an 8 bit type definition, consisting of a 4 bit tag specifying the item function, a 2 bit type, and a 2 bit size indicating the number of bytes of related data. All report descriptor data are represented in little endian format. If you need several different types of reports, you also need to include a report ID for each of them. See the HID specification for more on this subject.

Item type (bTag bType)	Value size (bSize)	Value (data)	Description
Usage page (0000 01)	10	Vendor defined page 1 (0xFF00)	Specify vendor usage page 1.
Usage (0000 10)	01	Vendor Usage 1 (0x01)	Specify vendor usage 1. As we have application specific interpretation of the reports, these usages are only for compatibility.
Collection (1010 00)	01	0 (0x00)	Application collection start
Logical minimum (0001 01)	01	0 (0x00)	Minimum logic value of report bytes
Logical maximum (0010 01)	01	255 (0xFF)	Maximum logic value of report bytes
Report size (0111 01)	01	8 (0x08)	8 bit report bytes
Report count (1001 01)	01	32 EP1_2_PACKET_SIZE(0x20)	Number of bytes in report
Input (1000 00)	01	Variable data (0000 0010)	Specify input report with previously defined parameters.
Usage (0000 10)	01	Vendor Usage 1 (0x01)	Specify vendor usage 1.
Output (1001 00)	01	Variable data (0000 0010)	Specify output report with previously defined parameters.
End collection (110000)	00	N/A	Application collection end

Table 4. Report descriptor

4.2 nRF24LU1+ application specific firmware

In our application the nRF24LU1+ works as a communication adapter between the USB interface of the computer, and the RF interface of the nRF24LE1. The functionality can be summed up with these points:

- Establishing USB communication with the computer, as described in the previous section
- Upon order from the host application, establishing an RF communication channel with the nRF24LE1
- Keeping the host up to date on the connection status with the remote device
- Forwarding selected messages from the host to the nRF24LE1, and relaying the corresponding responses back to the host

The specific application state machine mentioned in Figure 3. on page 7 for this application, and details concerning the RF communications with the nRF24LE1 are provided in the separate application note, nAN-18.

4.3 Host application: interaction with the USB HID library

4.3.1 Configuration

The host application interacts with a USB HID library which helps us configure and use the Windows HID drivers. The configuration process is illustrated with the sequence diagram of Figure 4. on page 12. We start with registering the VID and PID of our device, and subscribing to events which are fired when the device connects and disconnects, and data is sent and received. This setup procedure is done with an instance of the “UsbHidPort” class, which represents the USB HID Library. The “UsbHidPort” class is defined in the “USBHIDLib” namespace.

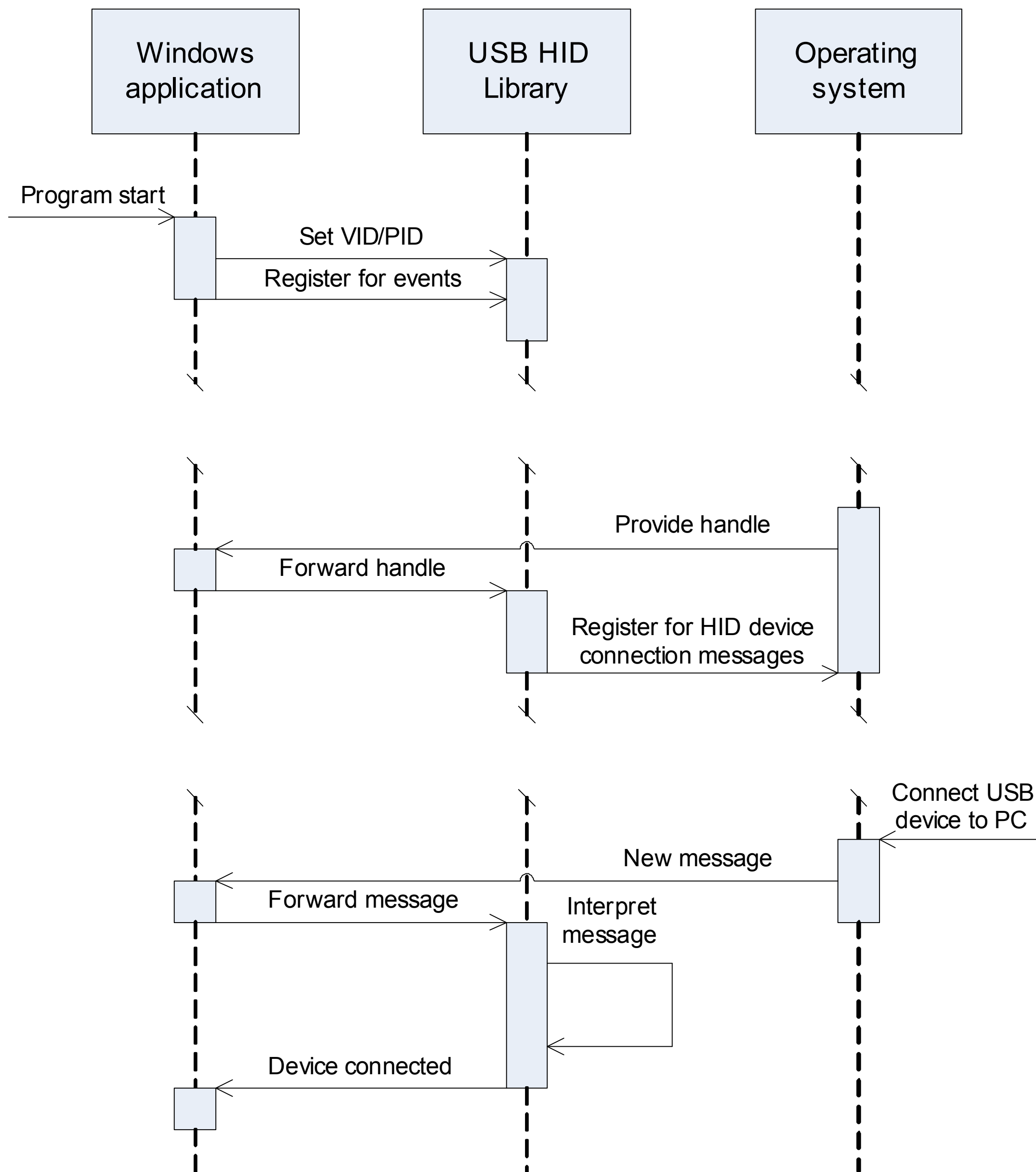

Figure 4. Host USB HID configuration steps

The USB HID library depends on the handle of our application, and its task is to forward device connection messages provided by Windows. The handle is the identifier for our form used by Windows, and its assignment happens at runtime. “OnHandleCreated” is a standard method called upon its creation that we must implement. In it, we must forward the handle to the “UsbHidPort” instance, so that it can make sure we receive notifications when Windows detects new HID devices.

The notification messages are provided to us through another standard method, “WndProc“. We forward these as well, leaving their interpretation to the USB HID library which notifies the application if the message was related to our specific device.

Both of these methods are found in our main form “nRFupdateForm”, and you should copy them into your own main Form when creating another Windows forms application.

4.3.2 Sending data to nRF24LU1+

Sending data is done by calling the method “UsbHidPort.SpecifiedDevice.SendData(transferData)”.

“UsbHidPort” is replaced by the name of the class instance. “transferData” is a byte array within the bounds of our specified packet size (defined by the report descriptor of nRF24LU1+).

In our implementation we use the function SendDataAndWait, which wraps the send function call together with a waiting loop exiting when the transfer is reported as complete.

4.3.3 Receiving data from nRF24LU1+

When data is received, the “OnDataRecieved” event of the “UsbHidPort” class is triggered. The received data is passed as a custom event argument class called “DataReceivedEventArgs”, which is defined in the “USBHIDLib” namespace. This class just contains a byte array with the length of the input report. We must add a new “DataReceivedEventHandler” and in the provided method decide what to do with the received data. In our solution this method is called “usb_OnDataRecieved”.

4.4 Host application: overview

As mentioned, our example application is the nRF Update firmware updater. Here is a quick summary of its most important features before we go into some more detail:

- Graphical user interface with intuitive controls and responsiveness to lower level events.
- Loading and verification of Keil compiled HEX firmware files targeted at the nRF24LE1.
- Line by line firmware transaction and verification ensuring a safe update together with the sophisticated update firmware on the nRF24LE1.

4.4.1 Main form

The main window, or form, of our application can be seen in Figure 5. Below it you will find a list explaining the different fields and buttons.

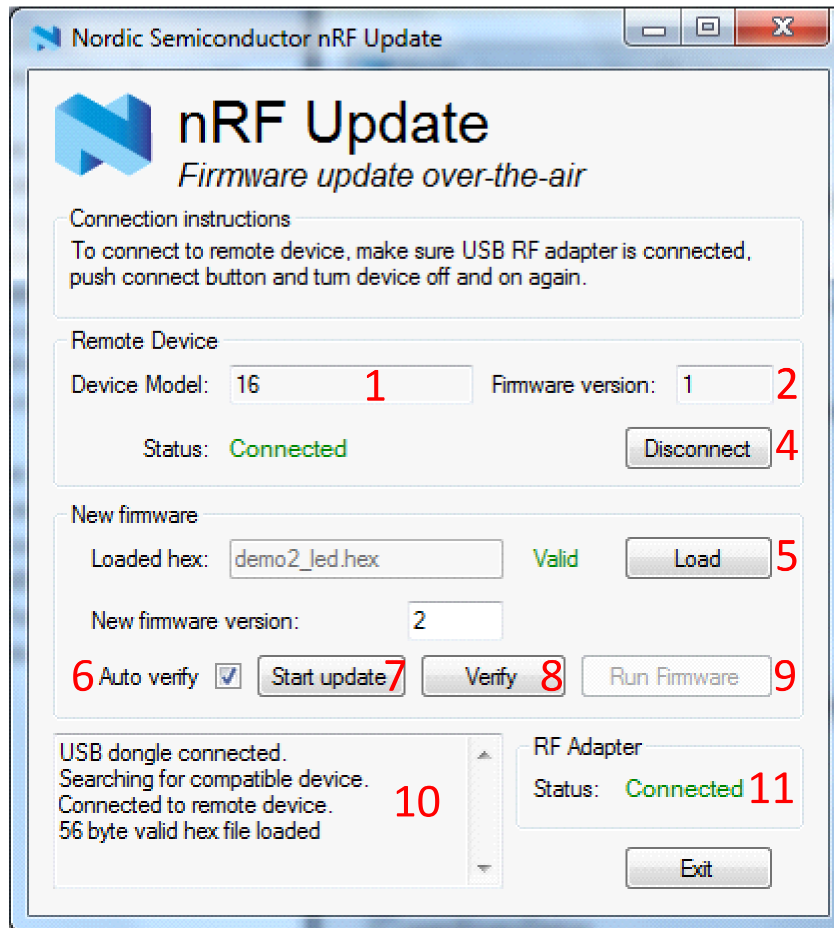

Figure 5. Main form

1. Field displaying a received number indicating what kind of supported device we have connected to. Might be replaced by an enumerator to display a more descriptive name.
2. Field displaying a number indicating the version of the currently running firmware on the remote RF device. N/A if the device reports no valid firmware.
3. Status label indicating whether we are connected, disconnected, or searching for a remote RF device.
4. Button with different functionality depending on connection status. Initiates search for remote device if currently disconnected, aborts a search in progress if searching, and disconnects with the device if we are currently connected.
5. Button opening a file system browser for selection of a HEX file to load. When a HEX file is selected, a file verification procedure is automatically run, checking for valid HEX symbols, format and checksums. The result of the file verification procedure is shown as either Valid or Invalid, and in the case of an invalid HEX file, a more detailed error is displayed in the status textbox.
6. Checkbox for automatic verification. If checked, the program will automatically initiate verification when the firmware update is completed.

7. Button initiating a line by line transaction of the selected HEX file. Enabled when a valid HEX file is loaded and we have a connection with a remote device.
8. Button initiating verification of the currently installed firmware on the remote device. Based on the currently loaded HEX file, each line is individually fetched and compared to check for errors.
9. Button for running the firmware after a successful firmware update. The functionality is identical to that of the disconnect button, and it is only included for clarity.
10. Textbox for detailed status messages.
11. Status label for the USB RF Adapter. Either connected or disconnected.

4.4.2 Progress window

Upon initiation of a firmware update or verification, a progress form is displayed, featuring a progress bar and a combined cancel and close button, as seen in Figure 6. The cancel functionality is enabled while a transaction is in progress, while the close functionality is enabled upon completion or cancelling of a transaction.

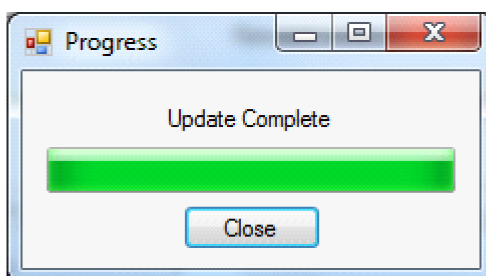

Figure 6. Progress form

4.5 Host application structure

The two main classes of our application are the GUI class “nRFupdateForm” and the controller class “nRF updateControl”, shown in Figure 7. Basically, the Form communicates with the controller by calling its methods, while the controller keeps the form up to date by raising events.

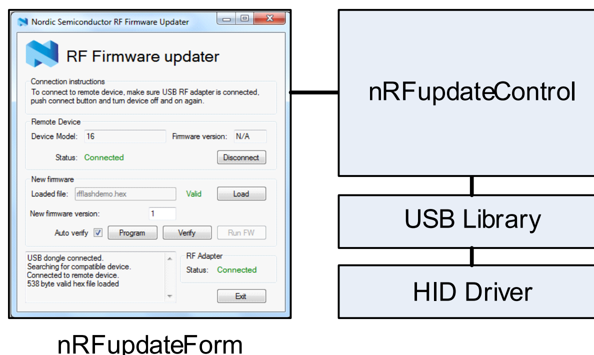

Figure 7. nRF firmware updater

The responsibilities of the “nRFupdateForm” class is limited to creating an intuitive presentation of the lower level events, and should not require further explanation beyond the comments found in its source code.

The “nRFupdateControl” class contains our application specific program logic, and is responsible for file I/O and communication with the nRF24LU1+ through the USB library. Most of the communication with the

nRF24LU1+ is just passed on to the nRF24LE1, so we will in the context of explaining the application consider all communication to happen between it and the nRF24LE1, unless explicitly noted otherwise.

4.5.1 Communication

All communication is initiated with a command from the host application, and concluded with a response from the nRF24LE1 in the form of an ACK or a NACK. The only exception is an unrequested NACK which is sent by the nRF24LU1+ when it unintentionally loses its connection with the nRF24LE1. If the sent command implies the return of data, this will be attached as payload to the corresponding ACK or NACK.

As the meaning of an ACK or a NACK is determined by the preceding command, we keep a variable called “expectedAck” aiding us in deciding what to do when one is received. When a USB message is received we call an ACK or a NACK handler, correspondingly, which in turn determines what to do depending on “expectedAck”.

4.5.2 Loading a HEX file

When a HEX file is selected in the GUI, we first load its contents into a character array, before it is parsed into individual binary HEX lines. The HEX lines’ length and checksum are calculated and compared with its respective fields, and the line lengths are summed so we know the total binary size of the firmware. When the HEX line with the reset vector is found (address zero), it is stored so we later may send it as part of the update start command.

4.5.3 Transferring the loaded firmware

The firmware transaction is initiated by sending the previously determined size and reset vector as well as a version optionally input in the main window. Also, a checksum for this initiation packet is calculated and provided. For more details on the communication protocol, see the application note nAN-18.

4.5.4 Verifying the remote firmware

The verification procedure uses the loaded HEX file to request specific addresses based on its HEX lines. If the received data does not match the stored HEX line, it is re-requested once before the verification procedure reports a defect remote firmware.

5 References

Visit the link www.usb.org/developers/hidpage.html for a USB HID specification, HID-usages tables, and HID report descriptor tool.

For the USB specification, refer to www.usb.org/developers/docs.html.

A good introduction to the USB standard can be found at http://www.beyondlogic.org/usbnutshell/usb1.shtml.

Liability disclaimer

Nordic Semiconductor ASA reserves the right to make changes without further notice to the product to improve reliability, function or design. Nordic Semiconductor ASA does not assume any liability arising out of the application or use of any product or circuits described herein.

Life support applications

Nordic Semiconductor's products are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. Nordic Semiconductor ASA customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Nordic Semiconductor ASA for any damages resulting from such improper use or sale.

Contact details

For your nearest dealer, please see http://www.nordicsemi.com

Receive available updates automatically by subscribing to eNews from our homepage or check our website regularly for any available updates.

Main office:

Otto Nielsens veg 12
7004 Trondheim
Phone: +47 72 89 89 00
Fax: +47 72 89 89 89
www.nordicsemi.com

Revision History

Date	Version	Description
September 2011	1.0	