

# **Application Note**

## **32-bit Cortex™-M0 MCU NuMicro® Family**

***An Example of CCID (Circuit Card Interface Devices)***

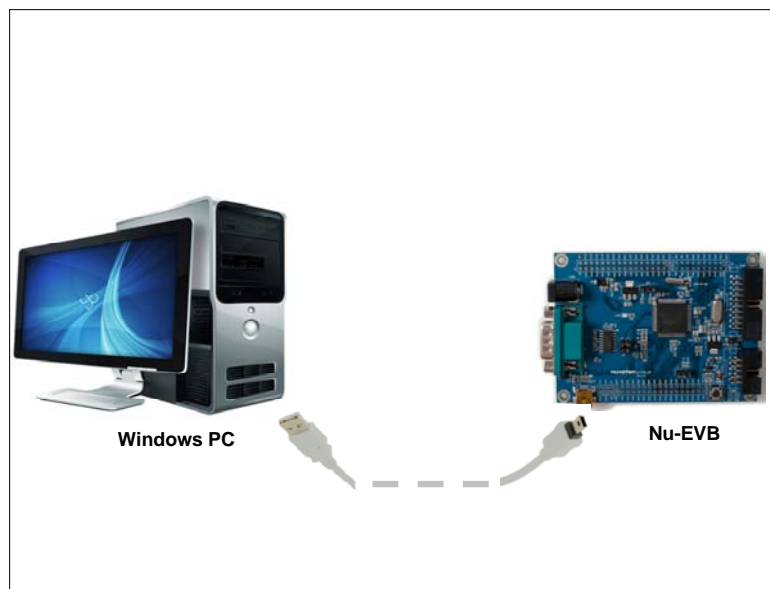
**Table of Contents-**

1	INTRODUCTION.....	2
2	CCID PROGRAM.....	3
2.1	System Initialization .....	5
2.2	USB Device Initialization.....	5
2.2.1	Endpoint Initialization.....	5
2.2.2	Descriptor Initialization .....	6
2.3	CCID Class Standard Request and Callback Function .....	6
2.3.1	“Get Clock Frequencies” and “Get Data Rates” through Control IN .....	7
2.3.2	“Abort” through Control.....	7
2.3.3	Command Message through Bulk OUT.....	8
2.3.4	Response Message through Bulk IN .....	9
2.3.5	Card Status through Interrupt IN .....	10
3	CUSTOMIZATION .....	12
3.1	VID (Vendor ID) / PID (Product ID).....	12
3.2	Standard Descriptors .....	12
3.2.1	Device descriptor.....	12
3.2.2	Configuration descriptor .....	13
3.2.3	String descriptors.....	15
3.3	Functions .....	15
3.3.1	Dispatch Message Function .....	15
4	PROGRAM EXECUTION.....	18
4.1	Source Directory .....	18
4.2	Execution Result.....	19
5	REVISION HISTORY .....	20

## 1 INTRODUCTION

The portable ATM is more popular now. Therefore the smart card reader is more important for user. Microsoft provides the driver inside the Windows OS, which follow the USB CCID (Circuits Card Interface Devices) class specification. And the smart card should follow the ISO/IEC 7816 specifications.

Nuvoton provides an example to demonstrate the capability of NuMicro™ family MCU to be a smart card reader. The following diagram illustrates the environment that runs the Smart Card Reader on a Nuvoton evaluation board, Nu-EVB.



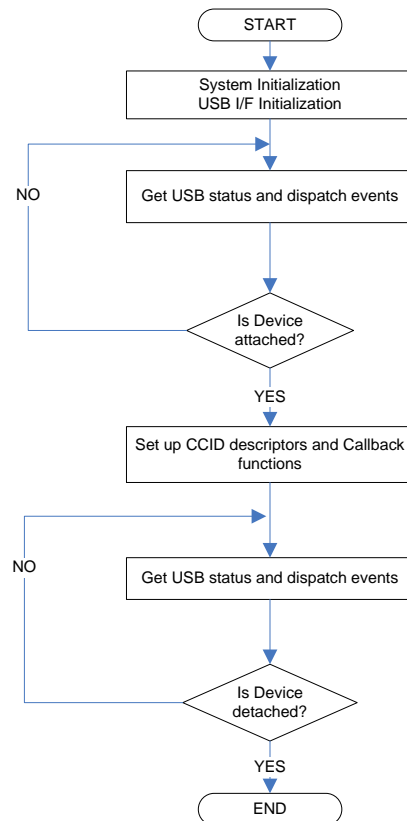
**Figure 1-1. An example of HID touch digitizer**

The Nu-EVB will be recognized as a smart card reader as soon as it is attached to PC's USB port. To emulate card reader function, program will return card not present to host. Because no smart card controller function is included, the host will recognize a smart card reader and wait for the card insert.

Nuvoton provides the source code to those users who want to develop a CCID smart card reader. The source code includes data structure of USB descriptors, functions of handling USB handshake protocol. User can customize her smart card device by modifying the source code. For example, USB Vendor ID/Product ID, or standard descriptors, or device class descriptor. This example doesn't contain the smart card controller function; instead Nuvoton provides UART library functions and Smart Card library functions to implement data transfer between smart card and NuMicro™ MCUs. By using dispatch function included in source code, smart card data could be sent to/ receive from host.

## 2 CCID PROGRAM

This example emulates a CCID card reader that is waiting for the smart card insert. The system clock, UART and USB Device are initialized firstly. Once attached to Host, a sequence of USB transfers will be handled properly that it will be recognized as a CCID smart card reader. Host will send commands and get response through USB request. A library function *DrvUSB\_DispatchEvent()* is used to check USB bus and endpoint status and call corresponding callback function to handle the event. Besides, a user-defined callback function *BulkOutData()* is used to parse the CCID command. After dispatch the CCID command, it will respond the message whenever a bulk IN transaction is received from HOST.



**Figure 2-1. Program Control Flow**

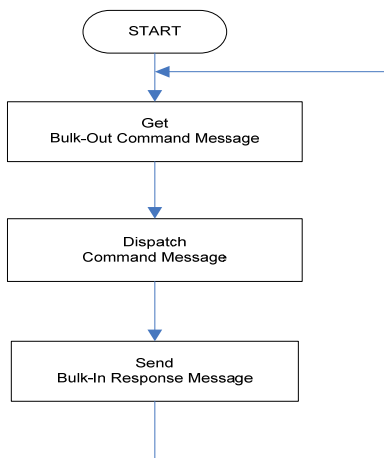


Figure 2-2. Bulk-OUT/IN Program Control Flow

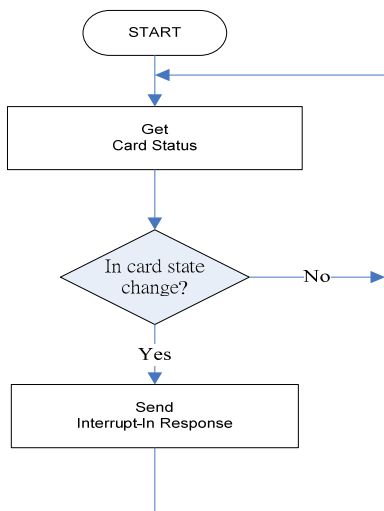


Figure 2-3. Interrupt-IN Program Control Flow

## 2.1 System Initialization

At system initialization phase, UART is configured as (115200, 8, n, 1) to be a debug message output interface. And the system clock is set to be 48MHz for running USB device function.

## 2.2 USB Device Initialization

The USBD initialization enables USB device hardware function and handles the USB events. It will initialize necessary endpoints, descriptors and installed callback functions.

### 2.2.1 Endpoint Initialization

This example uses five pipes including Control IN / OUT, Bulk IN / OUT and Interrupt IN. The Control pipe is used to handle “Abort”, “Get\_Clock\_Frequencies”, and “Get\_Data\_Rates” Class request. The Bulk Out pipe is used to handle command messages. The Bulk IN pipe is used to send response messages. The Interrupt IN pipe is used to send card status.

The *DrvUSB\_Open()* function is used to initialize endpoints information structures including endpoint number, maximum packet size, endpoint type, etc. The endpoint information structure is saved in a global array *sEpDescription[]* (defined at *Ccid\_Sys.c*). The following lists the structure content.

```
S_DRVUSB_EP_CTRL sEpDescription[] =
{
    /* EP Id 0, EP Addr 0, input, max packet size = 16 */
    {CTRL_EP_NUM      | EP_INPUT, CCID_MAX_PACKET_SIZE_CTRL, NULL},
    /* EP Id 1, EP Addr 0, output, max packet size = 16 */
    {CTRL_EP_NUM      | EP_OUTPUT, CCID_MAX_PACKET_SIZE_CTRL, NULL},
    /* EP Id 2, EP Addr 2, input, max packet size = 64 */
    {CCID_BULK_IN_EP_NUM | EP_INPUT,  CCID_MAX_PACKET_SIZE_BULK, NULL},
    /* EP Id 3, EP Addr 2, output, max packet size = 64 */
    {CCID_BULK_OUT_EP_NUM | EP_OUTPUT, CCID_MAX_PACKET_SIZE_BULK, NULL},
    /* EP Id 4, EP Addr 3, input, max packet size = 16 */
    {CCID_INT_IN_EP_NUM  | EP_INPUT,  CCID_MAX_PACKET_SIZE_INT, NULL},
    /* EP Id 5, EP Addr n, in/out, max packet size */
    {0x00, 0, NULL}
};
```

In addition to endpoint structures, a set of callback functions are required to process endpoint events. When *DrvUSB\_DispatchEvent()* function is called, it will check endpoint status and call corresponding callback function to handle USB event. The callback functions are saved in a global array *g\_sUsbOps[]* (defined at *Ccid\_Sys.c*). The following lists the array content.

```
S_DRVUSB_EVENT_PROCESS g_sUsbOps[12] =
{
    /* ctrl pipe0 (EP address 0) In ACK callback */
    {DrvUSB_CtrlDataInAck, &g_CCID_sDevice},
    /* ctrl pipe0 (EP address 0) Out ACK callback */
    {DrvUSB_CtrlDataOutAck, &g_CCID_sDevice},
```

```

/* EP address 1 In ACK callback */
{NULL, NULL},
/* EP address 1 Out ACK callback */
{NULL, NULL},
/* EP address 2 In ACK callback */
{CCID_BulkInCallback, &g_CCID_sDevice},
/* EP address 2 Out ACK callback */
{CCID_BulkOutCallback, &g_CCID_sDevice},
/* EP address 3 In ACK callback */
{CCID_IntInCallback, &g_CCID_sDevice},
/* EP address 3 Out ACK callback */
{NULL, NULL},
/* EP address 4 In ACK callback */
{NULL, NULL},
/* EP address 4 Out ACK callback */
{NULL, NULL},
/* EP address 5 In ACK callback */
{NULL, NULL},
/* EP address 5 Out ACK callback */
{NULL, NULL},
};

```

### 2.2.2 Descriptor Initialization

The *CCID\_Init()* function (defined at *CCID\_API.c*) is used to setup all descriptors including device, configuration and CCID class device descriptors. The input parameter of *CCID\_Init()* is a user-defined callback function - *BulkOutData()*. This callback function will be registered to USB device library to handle Bulk OUT command messages.

### 2.3 CCID Class Standard Request and Callback Function

This example is able to handle three CCID Class standard requests, including “Abort”, “Get\_ClockFrequencies” and “Get\_DataRates”. These Class-specific requests allow the host to abort the response portion of a command / response message pair and report a list of selectable clock frequencies and data rates.

Two callback functions, *CCID\_ClassCtrlAbortRequest()* and *CCID\_ClassCtrlRequest()* (defined at *CCID\_Sys.c*), are used to handle requests from Control IN/OUT pipes. These functions are registered to USB device driver via the following data structure.

```

S_DRVUSB_CTRL_CALLBACK_ENTRY g_asCtrlCallbackEntry[64] =
{
    //request type,command,etup ack handler, in ack handler,out ack handler, parameter
    {REQ_CLASS, CCID_Abort, CCID_ClassCtrlAbortRequest, 0, 0, &g_CCID_sDevice},
    {REQ_CLASS, CCID_GET_CLOCK_FREQUENCIES, CCID_ClassCtrlRequest,
    CCID_ClassCtrlRequestIn, 0, &g_CCID_sDevice},
    {REQ_CLASS, CCID_GET_DATA_RATES, CCID_ClassCtrlRequest,
    CCID_ClassCtrlRequestIn, 0, &g_CCID_sDevice},
};

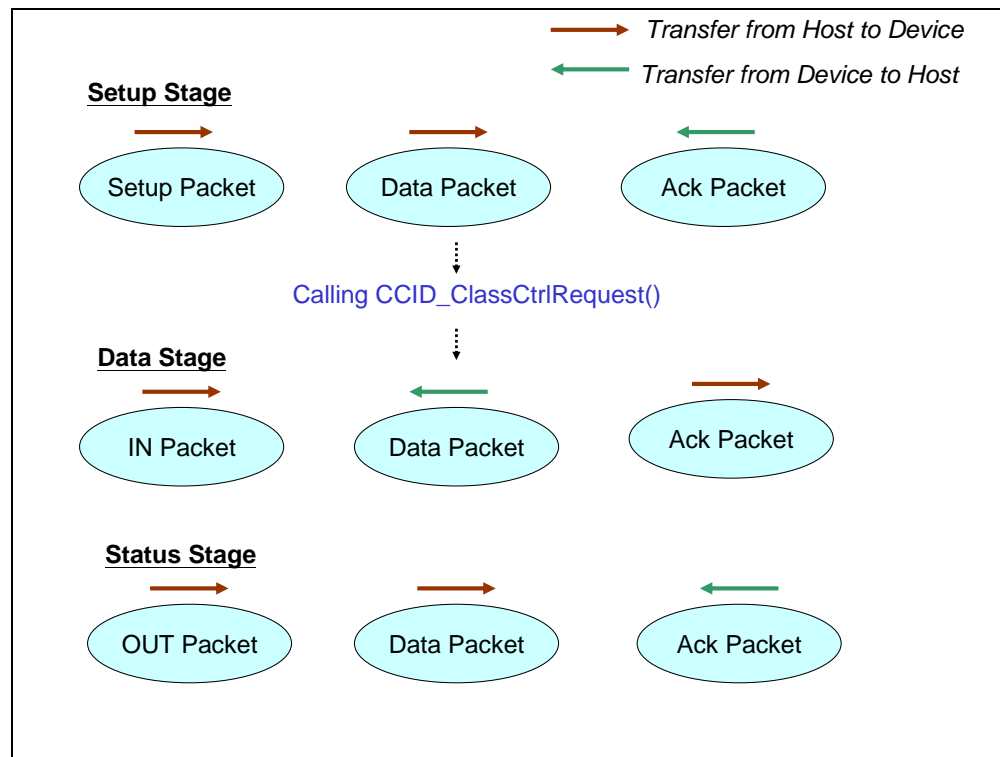
```

*Application Note*

For CCID device, host sends class command “Abort” to abort the response portion of a command / response message pair. This may be necessary to recover from error conditions and put the CCID into a state where it can receive a new command message. When a request command is received in an endpoint, its corresponding callback function will be called. The callback function has to prepare the report data that will be sent to host. The following sections describe the callback functions used for different requests.

**2.3.1 “Get Clock Frequencies” and “Get Data Rates” through Control IN**

When host wants to get clock frequency or data rates, it will use Control IN pipe to get the list report. Host firstly sends the Setup packet to device. USB device driver will acknowledge the Setup packet and call *CCID\_ClassCtrlRequest()* which is Control IN callback function. The clock frequency or data rate list report data is prepared in the callback function and will be sent to host in next Data Stage.



**Figure 2-4. Operation of USB Control IN transfer**

**2.3.2 “Abort” through Control**

For the Abort, it could be sent to device through Control pipe.

When the host wants to abort the message response, it will use Control pipe to send it. Host firstly sends the Setup packet to device. USB device driver will acknowledge the Setup packet by calling *CCID\_ClassCtrlAbortRequest()* which is Control Out callback function. Then send the abort command to smart card device.



## Application Note

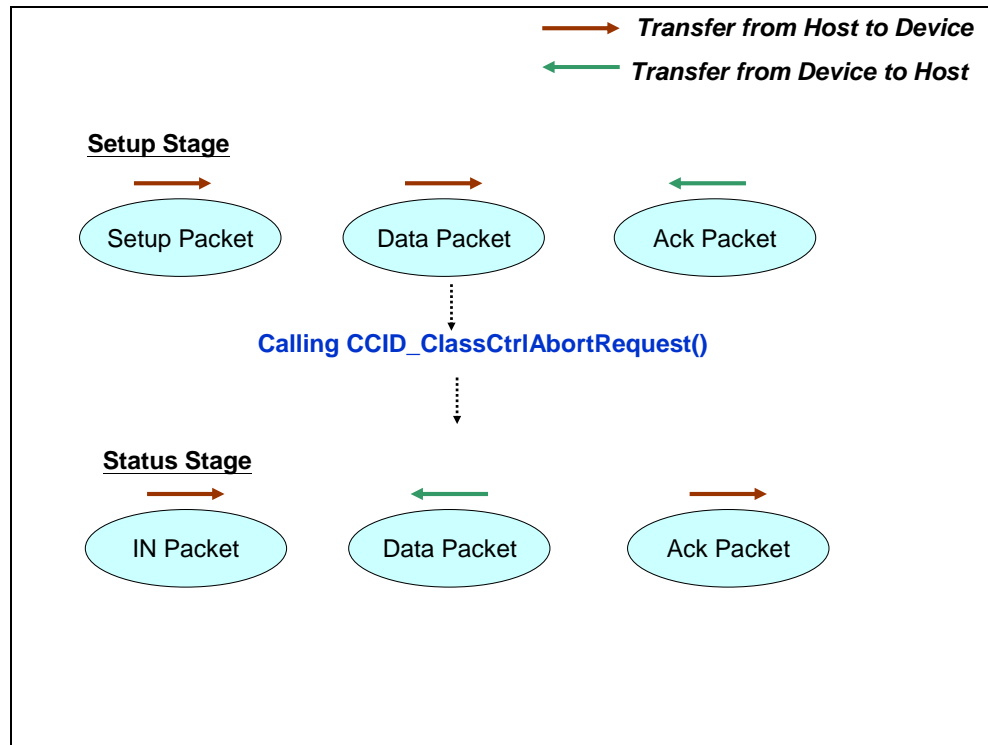


Figure 2-5. Operation of USB Control transfer

### 2.3.3 Command Message through Bulk OUT

Usually, the command message is sent from host through Bulk OUT pipe. Whenever a smart card insert, `BulkOutData()` callback function should be called to get the command from Bulk OUT buffer. USB device driver will parse it and dispatch command as soon as possible.

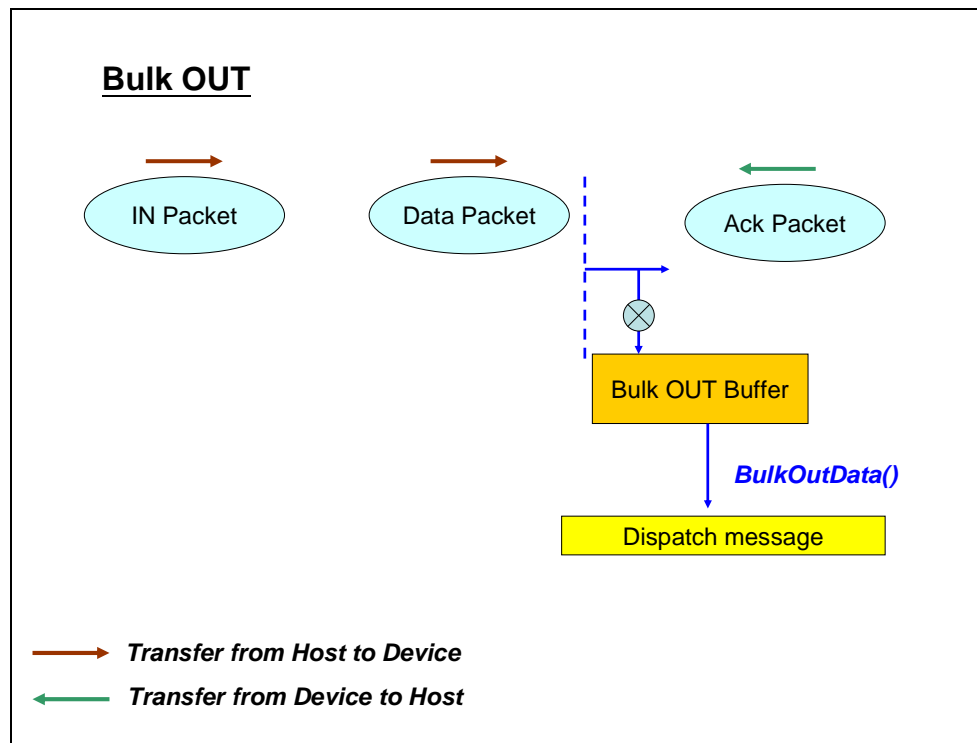
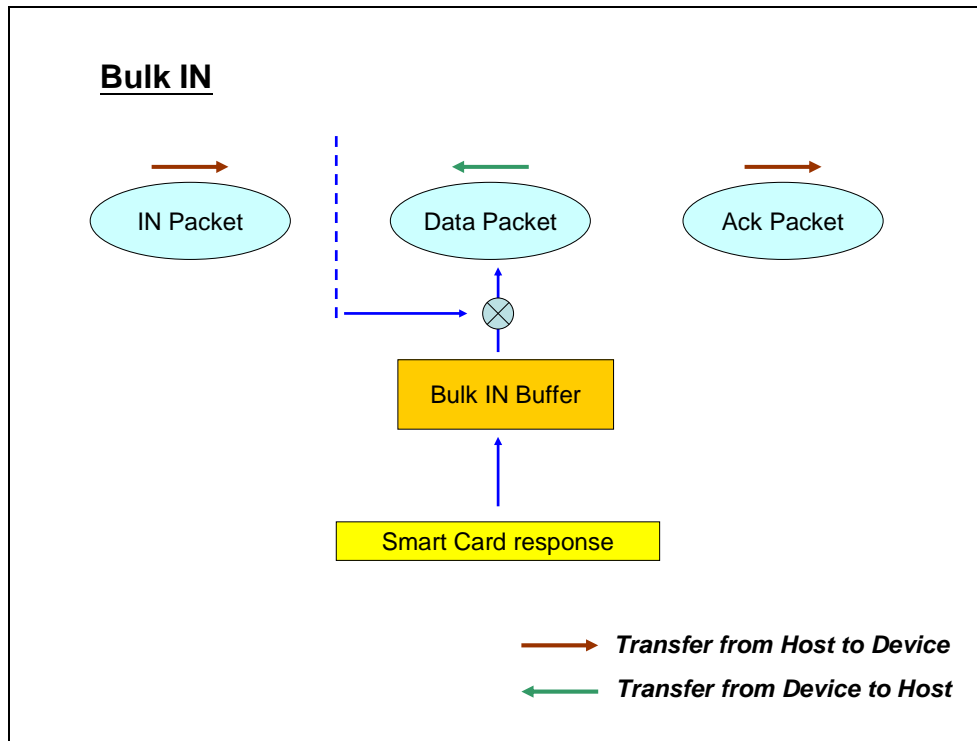


Figure 2-6. Operation of USB Bulk OUT transfer

### 2.3.4 Response Message through Bulk IN

After smart card is ready to response the command message, it will send the response message through Bulk IN pipe.



**Figure 2-7. Operation of USB Bulk IN transfer**

### 2.3.5 Card Status through Interrupt IN

Usually, the Input Report data is sent to host through Interrupt IN pipe. Host periodically gets the card status data. Whenever a card inserted / removed happens, the state should be written into Interrupt IN buffer. USB device driver will send it to host as soon as an IN token is received.

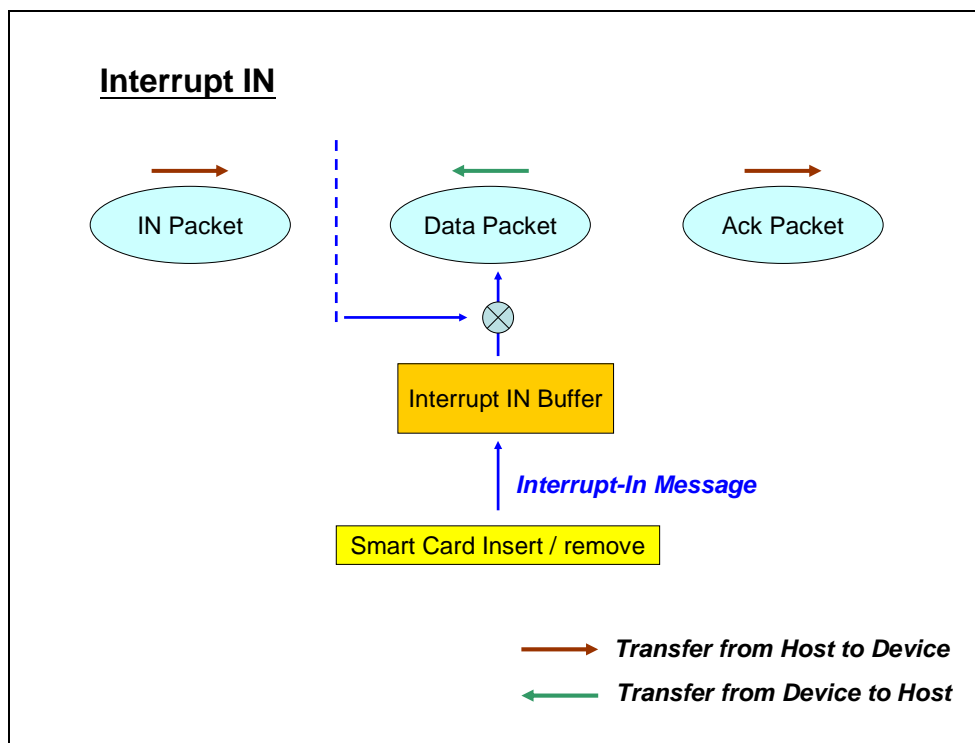


Figure 2-8. Operation of USB Interrupt IN transfer

### 3 CUSTOMIZATION

This example provides a framework of CCID smart card reader. User can change it according to her hardware functions. For example, the device ID, supported CCID commands, class device descriptor and so on. The data structure or function may need to be customized are listed as below.

#### 3.1 VID (Vendor ID) / PID (Product ID)

The default value of VID and PID are 0416H (Nuvoton) and C146h (NuMicro™ Family) respectively. User should modify it to fit her product.

Vendor ID	
Source	CCID_API.c
Type	Constant
Name	USB_VID

**Table 3-1. Vendor ID**

Product ID	
Source	CCID_API.c
Type	Constant
Name	USB_PID

**Table 3-2. Product ID**

#### 3.2 Standard Descriptors

The standard descriptors include Device, Configuration and String descriptors that must be supported in each USB device.

##### 3.2.1 Device descriptor

Device descriptor contains the VID/PID and serial number. The serial number is indicated by, gSerialIndex. If the CCID device has a unique serial number, this variable should be set to 03H. Otherwise, its value should be 00H.

Device Descriptor	
Source	CCID_API.c
Type	const uint8_t
Name	gau8DeviceDescriptor

**Table 3-3. Device Descriptor**

The content of device descriptor is listed at below.

## Application Note

```

#define USB_VID          0x0416
#define USB_PID          0xC143
#define gSerialIndex     0x00    // no serial number
// #define gSerialIndex  0x03    // has unique serial number
        :
        :
        :
const __align(4) uint8_t gau8DeviceDescriptor[] =
{
    LEN_DEVICE,          // bLength
    DESC_DEVICE,         // bDescriptorType
    0x10, 0x01,          // bcdUSB
    0x00,                // bDeviceClass
    0x00,                // bDeviceSubClass
    0x00,                // bDeviceProtocol
    CCID_MAX_PACKET_SIZE_CTRL, // bMaxPacketSize0
    USB_VID & 0x00FF, (USB_VID & 0xFF00) >> 8, // idVendor
    USB_PID & 0x00FF, (USB_PID & 0xFF00) >> 8, // idProduct
    0x00, 0x00,          // bcdDevice
    0x01,                // iManufacturer
    0x02,                // iProduct
    gSerialIndex,        // iSerialNumber
    0x01                // bNumConfigurations
};

```

### 3.2.2 Configuration descriptor

The configuration descriptor describes what endpoints are available. Each endpoint is specified in an endpoint descriptor. This example supports Bulk Out, Bulk IN and Interrupt IN endpoints. User can add other endpoint descriptor if necessary.

Configuration Descriptor	
Source	CCID_API.c
Type	const uint8_t
Name	gau8ConfigDescriptor

**Table 3-4. Configuration Descriptor**

The content of configuration descriptor is listed at below.

```

const __align(4) uint8_t gau8ConfigDescriptor[] =
{
    LEN_CONFIG,          // bLength
    DESC_CONFIG,         // bDescriptorType
    // wTotalLength
    (LEN_CONFIG+LEN_INTERFACE+LEN_CCID+LEN_ENDPOINT*3) & 0x00FF,
    ((LEN_CONFIG+LEN_INTERFACE+LEN_CCID+LEN_ENDPOINT*3) & 0xFF00) >> 8,
    0x01,                // bNumInterfaces
    0x01,                // bConfigurationValue

```

## Application Note

```

0x00,          // iConfiguration
0xC0,          // bmAttributes
0x32,          // MaxPower
// Interface descriptor (Interface 0 = Smart Card Reader)
LEN_INTERFACE, // bLength
DESC_INTERFACE, // bDescriptorType
0x00,          // bInterfaceNumber
0x00,          // bAlternateSetting
0x03,         // bNumEndpoints
0x0B,          // bInterfaceClass
0x00,          // bInterfaceSubClass
0x00,          // bInterfaceProtocol
0x00,          // iInterface
// CCID class descriptor
0x36,          // bLength: CCID Descriptor size
0x21,          // bDescriptorType: CCID specific number
0x10,          // bcdCCID(LSB): CCID Class Spec release number (1.10)
0x01,          // bcdCCID(MSB)
0x00,          // bMaxSlotIndex
0x07,          // bVoltageSupport: 5v, 3v and 1.8v
0x03,0x00,0x00,0x00, // dwProtocols: supports T=0 and T=1
0xA6,0x0E,0x00,0x00, // dwDefaultClock: 3.75MHz
0x4C,0x1D,0x00,0x00, // dwMaximumClock: 7.5MHz
0x00,          // bNumClockSupported => no manual setting
0x60,0x27,0x00,0x00, // dwDataRate: 10080 bps
0xB4,0xC4,0x04,0x00, // dwMaxDataRate: 312500 bps
0x00,          // bNumDataRatesSupported => no manual setting
0xFE,0x00,0x00,0x00, /* dwMaxIFSD: 0 (T=0 only) */
0x00,0x00,0x00,0x00, /* dwSynchProtocols */
0x00,0x00,0x00,0x00, /* dwMechanical: no special characteristics */
0x30,0x00,0x01,0x00, // dwFeatures: clk, baud rate, voltage : automatic
// CCID can set ICC in clock stop mode.
0x0F,0x01,0x00,0x00, /* dwMaxCCIDMessageLength : Maximun block size header*/
/* 261 + 10 */
0x00,          /* bClassGetResponse*/
0x00,          /* bClassEnvelope */
0x00,0x00,     /* wLcdLayout */
0x00,          /* bPINSupport : no PIN verif and modif */
0x01,          /* bMaxCCIDBusySlots */
// Endpoint 1 descriptor (Interrupt in)
LEN_ENDPOINT, // bLength
DESC_ENDPOINT, // bDescriptorType
CCID_INT_IN_EP_NUM | 0x80, // bEndpointAddress
EP_INT,       // bmAttributes
CCID_MAX_PACKET_SIZE_INT & 0x00FF, // wMaxPacketSize
(CCID_MAX_PACKET_SIZE_INT & 0xFF00) >> 8,
CCID_DEFAULT_INTERVAL_INT, // bInterval

// Endpoint 2 descriptor (Bulk out)
LEN_ENDPOINT, // bLength
DESC_ENDPOINT, // bDescriptorType
CCID_BULK_OUT_EP_NUM, // bEndpointAddress

```

## Application Note

```

EP_BULK,                                // bmAttributes
CCID_MAX_PACKET_SIZE_BULK & 0x00FF,      // wMaxPacketSize
(CCID_MAX_PACKET_SIZE_BULK & 0xFF00) >> 8,
CCID_DEFAULT_INTERVAL_BULK,              // bInterval

// Endpoint 2 descriptor (Bulk in)
LEN_ENDPOINT,                            // bLength
DESC_ENDPOINT,                           // bDescriptorType
CCID_BULK_IN_EP_NUM | 0x80,               // bEndpointAddress
EP_BULK,                                 // bmAttributes
CCID_MAX_PACKET_SIZE_BULK & 0x00FF,      // wMaxPacketSize
(CCID_MAX_PACKET_SIZE_BULK & 0xFF00) >> 8,
CCID_DEFAULT_INTERVAL_BULK               // bInterval
/* Add others endpoint descriptors */
...
};

```

### 3.2.3 String descriptors

The string descriptors provide device information including vendor name, product name and serial number.

String Descriptors	
Source	CCID_API.c
Type	const uint8_t
Name	gau8VendorStringDescriptor gau8ProductStringDescriptor gau8StringSerial

**Table 3-5. String Descriptors**

## 3.3 Functions

The function is used to handle the card command / response message.

### 3.3.1 Dispatch Message Function

The host sent the command to smart card through Bulk OUT pipe. Whenever a command is received, *CCID\_DispatchMessage()* function should be called to send message to smart card. And the smart card response message will be sent to host after received bulk IN token.

DispatchMessage	
Source	CCID_API.c
Syntax	void CCID_DispatchMessage(void)



Table 3-10. DispatchMessage Function

```

void CCID_DispatchMessage(void)
{
    uint8_t ErrorCode;

    if(g_CCID_sDevice.isBulkOutReady)
    {
        switch(UsbMessageBuffer[OFFSET_BMESSAGE_TYPE])
        {
            case PC_TO_RDR_ICCPOWERON:
                ErrorCode = PC_to_RDR_IccPowerOn();
                RDR_to_PC_DataBlock(ErrorCode);
                break;

            case PC_TO_RDR_ICCPOWEROFF:
                ErrorCode = PC_to_RDR_IccPowerOff();
                RDR_to_PC_SlotStatus(ErrorCode);
                break;

            case PC_TO_RDR_GETSLOTSTATUS:
                ErrorCode = PC_to_RDR_GetSlotStatus();
                RDR_to_PC_SlotStatus(ErrorCode);
                break;

            case PC_TO_RDR_XFRBLOCK:
                ErrorCode = PC_to_RDR_XfrBlock();
                RDR_to_PC_DataBlock(ErrorCode);
                break;

            case PC_TO_RDR_GETPARAMETERS:
                ErrorCode = PC_to_RDR_GetParameters();
                RDR_to_PC_Parameters(ErrorCode);
                break;

            case PC_TO_RDR_RESETPARAMETERS:
                ErrorCode = PC_to_RDR_ResetParameters();
                RDR_to_PC_Parameters(ErrorCode);
                break;

            case PC_TO_RDR_SETPARAMETERS:
                ErrorCode = PC_to_RDR_SetParameters();
                RDR_to_PC_Parameters(ErrorCode);
                break;

            case PC_TO_RDR_ESCAPE:
                ErrorCode = PC_to_RDR_Escape();
                RDR_to_PC_Escape(ErrorCode);
                break;

            case PC_TO_RDR_ICCCLOCK:
                ErrorCode = PC_to_RDR_IccClock();
                RDR_to_PC_SlotStatus(ErrorCode);
                break;

            case PC_TO_RDR_ABORT:
                ErrorCode = PC_to_RDR_Abort();
                RDR_to_PC_SlotStatus(ErrorCode);
                break;
        }
    }
}

```

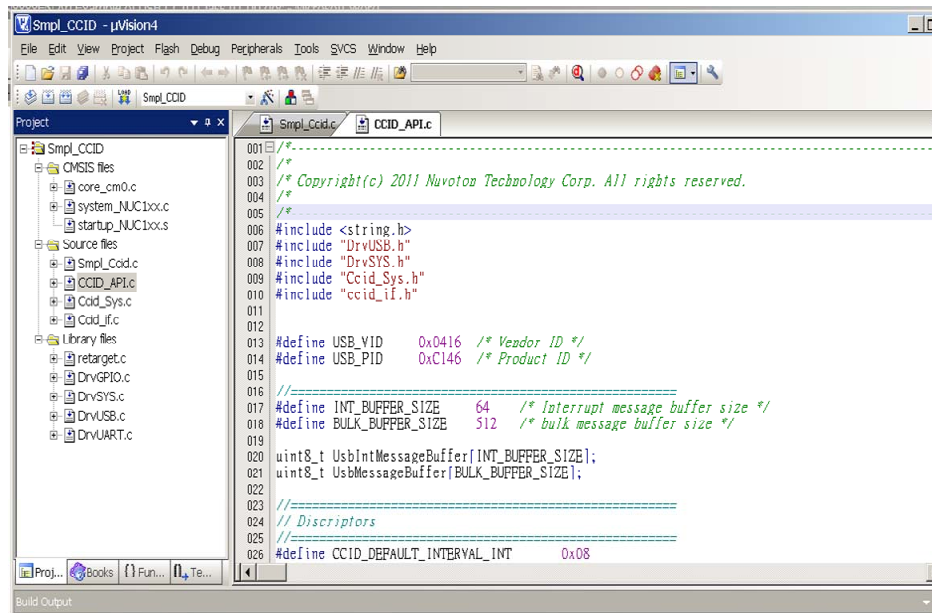
*Application Note*

```
        case PC_TO_RDR_SETDATARATEANDCLOCKFREQUENCY:
        case PC_TO_RDR_SECURE:
        case PC_TO_RDR_T0APDU:
        case PC_TO_RDR_MECHANICAL:
        default:
            CmdNotSupported();
            break;
    }
    UsbBulkInMessage();
    g_CCID_sDevice.isBulkOutReady = 0;
}
}
```

## 4 PROGRAM EXECUTION

### 4.1 Source Directory

This example code is created by using Keil uVision v4.03. A Smpl\_CCID project file is saved at the top directory of source code. After open this project file, the following working window will be displayed.



**Figure 4-1. Snapshot of Keil uVision Working Window**

The project includes the example source files and NuMicro™ BSP files (CMSIS and device driver files). There're three example source files.

- **Smpl\_CCID.c**  
This file contains *main()* function that does system initialization.
- **CCID\_API.c**  
This file includes global variables of descriptors, major operating function *CCID\_MainProcess()*, and dispatch message function *CCID\_DispatchMessage()*.
- **CCID\_Sys.c**  
This file includes those functions of setting endpoint, endpoint callback functions and so on.
- **CCID\_if.c**  
This file includes those functions of card relative setting and so on.

## 4.2 Execution Result

The program should be run on a Nu-EVB. When the board is attached to Windows PC, it will be recognized as a CCID smart card reader device.

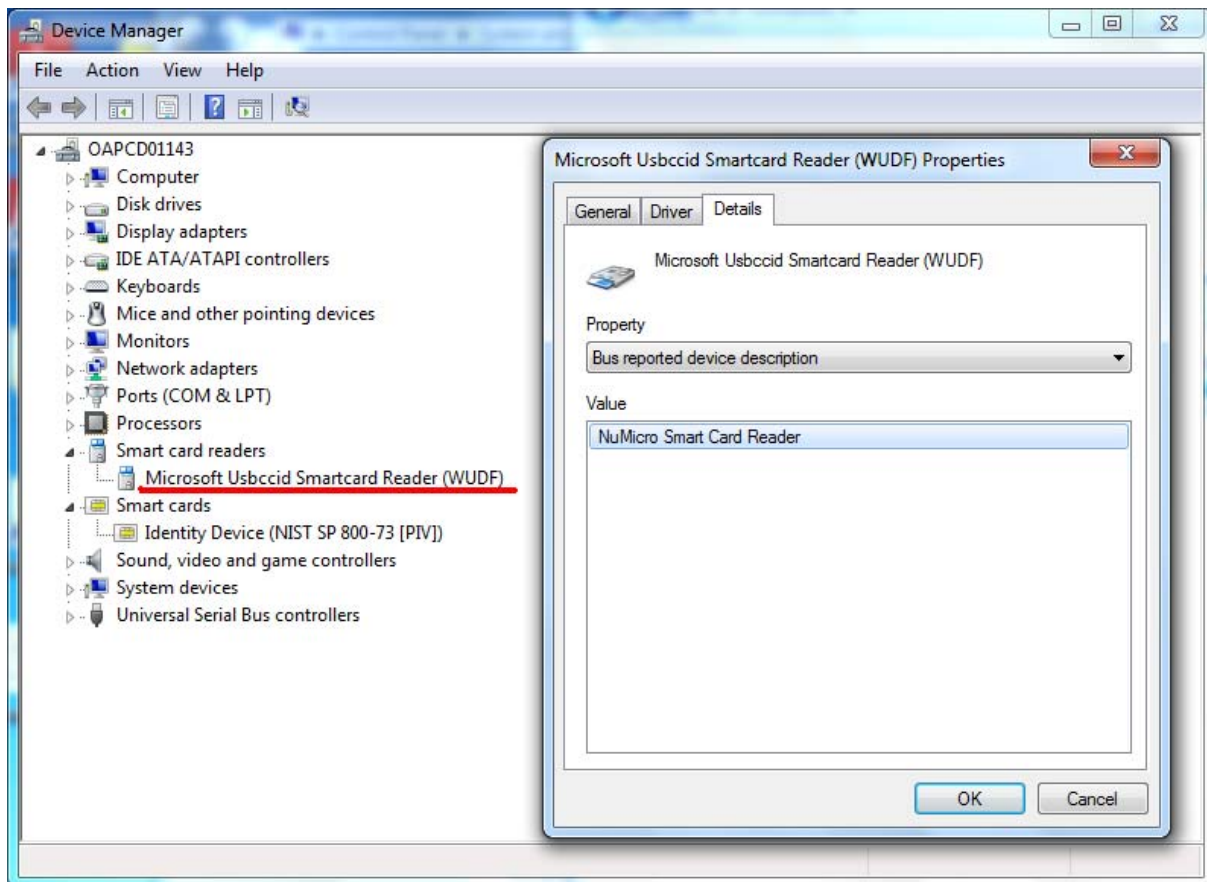


Figure 4-2. Snapshot of Windows Paint

## 5 REVISION HISTORY

REV.	DATE	DESCRIPTION
1.00	March 10, 2011	Created.

### **Important Notice**

**Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, “Insecure Usage”.**

**Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.**

**All Insecure Usage shall be made at customer’s risk, and in the event that third parties lay claims to Nuvoton as a result of customer’s Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.**

---

*Please note that all data and specifications are subject to change without notice.  
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.*