Roche Diagnostics Diabetes Care © 2004 All rights reserved	**SPEC USB-HID DEVICE COMM PROTOCOL VER 1.01**

Revisions *For Revision Approval Date See CN*

Rev	Chg. #	Description	Reviser	Chg. Date	Approver
B	DCMan4111	See CN DCMan4111	M. Young	01/26/06	B. Markisohn
C	DCMAN4461	Added config command and revised the HID Protocol Version number and modified as necessary to support ACUC	B. Markisohn	07/07/06	M. Young
D	DCMAN5902	Modifications to support ACUC	B. Markisohn	04/24/07	M. Young
E	RDODC5415	Correct numbering of steps Section 7.3.1.3	T. Oelmann	3/28/12	B. Markisohn

Roche Diagnostics

USB-HID Device Communication Specification Protocol Version 1.01

	Approvals	Date
Originator	**Wiliam Powell, Indesign**	**12/30/04**
Engineer	**Morris Young**	**08/03/05**
Engineer		
Approved	**Chris Baker**	**See CN**

Release Doc. No.
DCMAN3422

Ref RDC Part No.

Type	Document Number	Rev
SPEC	**7000804**	**E**

Table of Contents

Table Of Figures

Revision	Section	Description
B	5.1.1	In table 1, corrected idProduct
B	7.2.5	In the 'Format' table, description of p5 was corrected
B	7.2.5	In the 'Format' table, bytecnt was corrected
B	7.2.3	Count for 'CFG' was corrected
C	General	Changed SmartLink to RUSB except for command nanes
C	1.0	Added description and support for the ACUC
C	2.0	Added definition for ACUC and RUSB
C	3.1	Added references for SmartLink CoSo (SRD and SDD)
C	5.1.1	Added ACUC Device Descriptor table
C	5.2.2	Added ACUC Product ID String table
C	6.1	HID Report Modifications: • Reports associated with firmware upgraded tagged as optional and behavior when used with ACUC described. • Additional description added to LED and State commands for ACUC's limitations.
C	7.2.10	• Modified cfgRX_TIMEOUT tick size from 10 ms to 20 ms and the default to 300 ms • Added cfgTX_BYTE_TO_WAIT to control inter-character delay during transmission
D	Table 2	• ACUC Device Descriptor Offset 16 iSerialNumber Value 0x00 ; ACUC has no serial number
D	Table 3	• Configuration Descriptor Offset 2 wTotalLength Value 0x22
D	Table 4	• HID Interface Descriptor Offset 2 bInterfaceNumber Value 0x00 ; 0 for ACUC and 1 for Smart Pix
D	Table 5	• HID Descriptor Offset 7 wDescriptorLenth removed TBD and replaced value with 0x32
D	Table 6	• HID Endpoint Descriptor Offset 2 added to description note ; IN and endpoint #3
D	Table 11	• Serial Number String, Offset 2 bString removed Description Note
E	7.3.1.3	Corrected numbering of steps for Read Number of Records

1. Purpose

The purpose of this document is to describe the RUSBHuman Interface Device (HID) interface. RUSB is intended to be a universal infrared (IR) communication device capable of interacting with other IR devices. The RUSB HID interface allows an application residing on a Personal Computer (PC) to interact directly with an IR device. The document contains the definition of the three major components of the interface:

- HID descriptors
- HID Report descriptions
- Sequencing Engine API.

The USB HID descriptors define the device specific information necessary to properly identify the device to the host application as a general-purpose HID device. The HID Reports are the structures used to send and receive data across the USB. The sequencing engine manages the serial interactions with Devices by embedding the timing critical portions of the communication protocol within RUSB and removing any critical response timing from the PC or the USB communication controlling the RUSB.

In addition to Smart Pix, which is a composite device that implements both HID and mass storage USB classes, this document also supports the HID Interface for the Accu-Chek USB Cable (ACUC) that only implements a HID interface. Those USB HID commands that are unnecessary for a pure HID device have been marked as optional.

It is assumed that the Smart Pix device interface will only work with Microsoft Windows 2000 and above. The intended audience of this document is software engineers implementing the RUSB firmware and Test Engineers who will validate the implementation. An implicit knowledge of USB and specifically USB HID report structures is assumed.

2. Definitions

ACUC – Accu-Chek USB Cable.

API – Application Programming Interface.

bG – an abbreviation for blood glucose.

Communication Sequence – a command, its parameters, and the responses to that command sent from the host to an instrument.

Device – a meter or pump that the RUSB may communicate with via the IR link.

Field – a piece of data sent to or from a device. This data may contain multiple components.

HID – USB Human Interface Device.

Host – a computerized external device that can initiate communications and provide serial commands to an instrument. Certain instruments with DM capabilities can function as a host if connected to a device (e.g. bG meter).

Instrument – for purposes of this standard, an instrument is defined to be a battery-powered electronic device developed for sale to external customers. The above-mentioned instruments with DM capabilities function as instruments if they're connected to a host (e.g. PC).

Inter-Character Delay – a minimum time delay required in addition to any specified stop or start bits between serial characters being transmitted to an instrument.

IR – an abbreviation for infrared light, light in the non-visible spectrum. Instruments and hosts can use this light generated by electronic devices to send data bits to a host or instrument

LSB – an abbreviation for least significant bit. This refers to the bit with the smallest value (2^0) in a character.

Meter – analogous with bG meter.

Power Down – an instrument state where, although the instrument is turned off and not performing its intended function, it can be powered up either through user action or serial communications.

Pump – an insulin pump (e.g. Spirit, DTRON PLUS, etc)

Reset – the process of placing an instrument in a known, initialized state, whether initiated by application of power or by firmware.

RD – an abbreviation for Roche Diagnostics. The German located company is known as Roche Diagnostics Germany (RDG), the United States located company is known as Roche Diagnostics Operations (RD).

RUSB – Roche USB Device. Generic acronym for a Roche USB-IR device currently limited to Smart Pix and Accu-Chek USB Cable.

Serial – the process of sending data one bit at a time across a data communication link.

SL – Sequencing Language.

USB – Universal Serial Bus.

3. References

3.1 Internal References

1. Disetronic mini-TRON: Specification of communication commands, Version 2, Revision D
2. Disetronic N-TRON / DiaLog: Communication Protocol for N-TRON, Version V3.02 (for DiaLog for N-TRON)
3. Disentronic Gondor: Infrarot Kommunikationsprotokoll, Version 2, Revision D
4. Roche Diagnostics Instrument Communication Interface, (rev. 1.1), dated 1/17/2000
5. SRD Smart Link CoSo (rev. 1.0)]
6. SDD Smart Link CoSo (rev. 1.0)

3.2 External References

1. Universal Serial Bus Specification, Revision 1.1.
2. USB HID Usage Tables 1.11 (**www.usb.org/developers/hidpage**)
3. USB Device Class Definitions for HID Specification 1.1 (**www.usb.org/developers/hidpage**)

4. Simple USB - Sequence Engine – IR Device Example

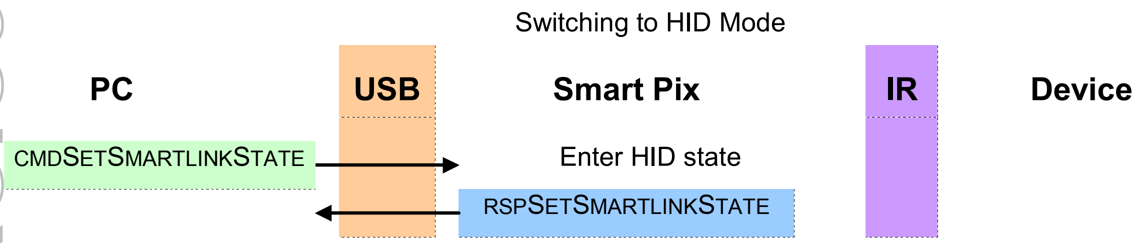
Sequences commands, as defined in section 7.2, are serial commands that instruct the sequence engine how to interact with an IR device. The HID reports, as defined in section 6.1, are the commands that transport the sequence commands to the sequencing engine then return any data back to the PC via the USB. Because the sequence size may exceed the HID packet size, the interface must be capable of reading or writing in blocks that can be reassembled to form the larger method.

Figure 1 shows how the PC will communicate with RUSB to accomplish an IR action. The PC is always the master in the communication, the RUSB is always the slave, and the communication is synchronous. In other words, the PC will send a command and get a response before another command is sent. The only exception to the command/response communication is the CMD_RESET, which can be used to place the USB-HID device into a known state if a response is never received (response timeout).

A sequence is a combination of Sequence API primitives (see Section 7) that when linked together perform a timing critical interaction with a device. The downloaded sequence data will typically hold many primitives and is downloaded as a contiguous block of data (broken up into the appropriate block size needed for the USB transfer).

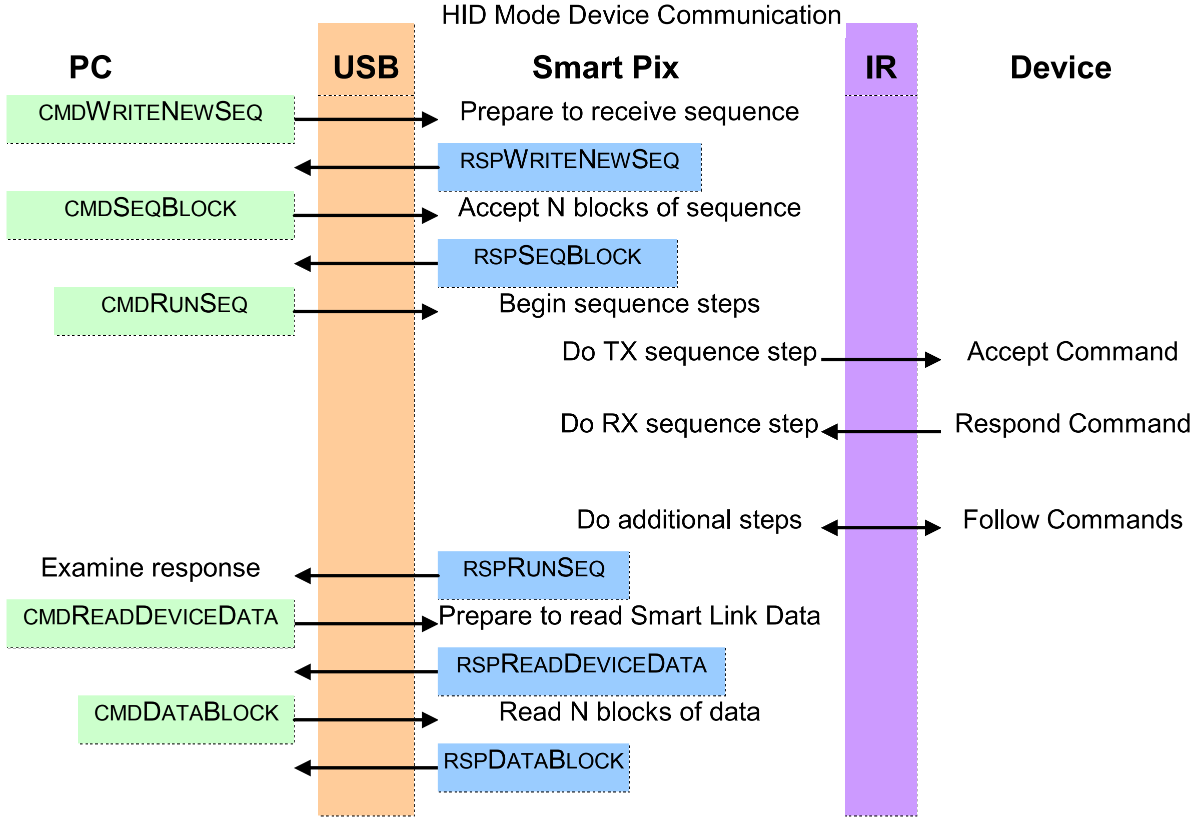
This diagram is a short synopsis of the process. More detailed examples can be found in section 7.3.

Figure 1: PC-to-Device Communication Example with Smart Pix

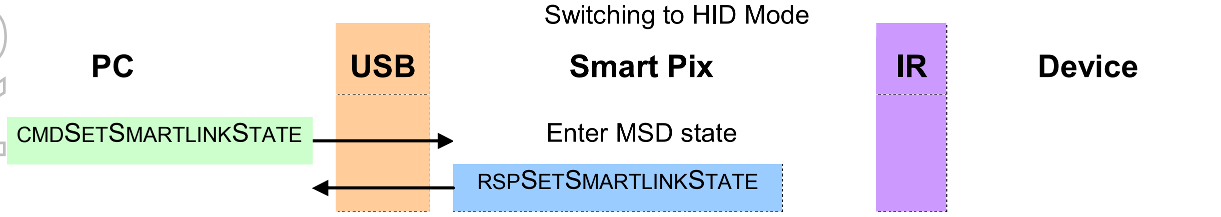


* Only needed if in MSD mode, may be rejected if MSD mode is busy.

HID Mode Device Communication



Switching to HID Mode



* HID mode will timeout to MSD mode or can be exited by setting the Smart Pix state to MSD.

The PC begins by sending CMDWRITENEWSEQUENCE to indicate that a new sequence is coming. The number of blocks to be sent, the byte count of the sequence, and the number of steps in the sequence are included in the command. The Smart Pix response (RSPWRITENEWSEQUENCE) will be a positive acknowledgement unless the device is already actively executing a previous sequence. If ACKed, the PC continues to send the sequence data using CMDSEQBLOCK to the Smart Pix one block at a time. The response for each block (RSPSEQBLOCK) will be ACK if received properly by the Smart Pix.

Once the new sequence has been successfully loaded into the Smart Pix, the sequence (IR communication with a device) is initiated by sending CMDRUNSEQ. The PC waits for the run response (RSPRUNSEQ) which is sent when the sequence of IR communication with the device

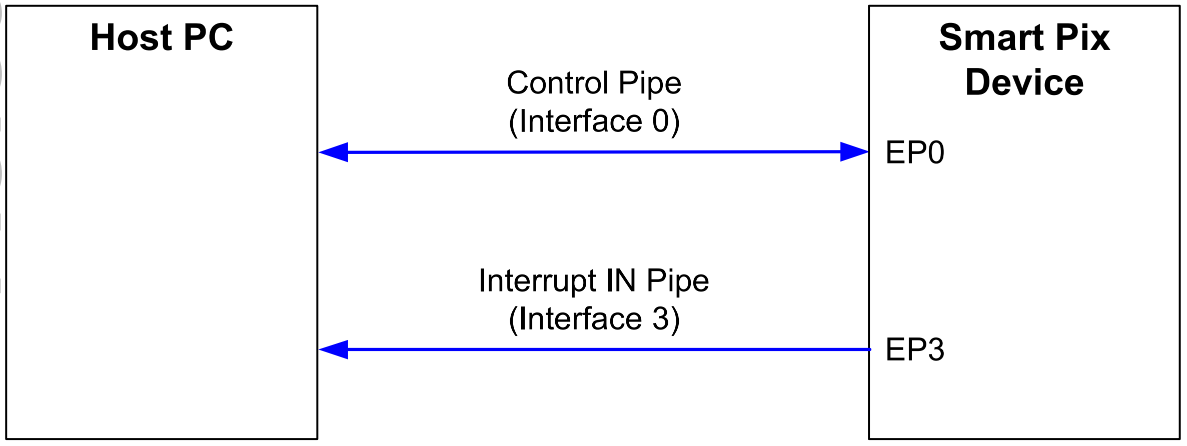
has completed. As the sequence executes, any bytes received from the device are collected in the response data buffer. The run response will contain a sequence status code, the ending sequence step, and the size of the device response data available to be read from the Smart Pix by the PC.
The PC can then chose to read the receive buffer by sending CMDREADDEVICEDATA, which prepares the Smart Pix for reading data. The receive buffer is read in blocks using the CMDDATABLOCK and RSPDATABLOCK commands.

In the event that a sequence command fails, the response (RSPRUNSEQ) will be an ACK (the sequence did complete) and the PC will need to interpret the sequence status code and ending step to determine where the failure occurred. The PC based application is responsible for managing the timeout scenarios between OUT and expected IN reports and for putting the device back into an idle state if an error occurs while communicating with the device.

5. HID Descriptors

This section defines the USB HID descriptors for the RUSB system. Included are the descriptors necessary to properly identify the device to the host application. Figure 2 shows a high-level view of the HID interface. Two endpoints are used for communication between the device and the host. The control endpoint handles the standard USB requests and the HID "OUT" reports. The interrupt pipe carries the HID "IN" reports. The interrupt pipe is polled on a regular basis by the host. It sends data only when necessary (to report a status change or respond to a HID "out" report).

Figure 2. HID Endpoint Configuration



5.1 Descriptor Definitions

The tables below describe the different descriptors reported by the USB interface. In the descriptors, any multiple byte values will be encoded in little endian format, e.g. a value of 0x1234 will be stored in memory as: location n = 0x34, location n+1 = 0x12. The number of bytes will be indicated by the number of digits in the number, e.g. 0x1234 would be a two byte value.

5.1.1 Device Descriptors

Table 1 Smart Pix Device Descriptor

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*blength*	*1*	*0x12*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x01*	*This descriptor type,* ***Device****.*
2	*bcdUSB*	*2*	*0x0110*	*USB specification version number,* ***Version 1.1****.*
4	*bDeviceClass*	*1*	*0x00*	*Class code.*
5	*bDeviceSubClass*	*1*	*0x00*	*Subclass code.*
6	*bDeviceProtocol*	*1*	*0x00*	*Protocol code.*
7	*bMaxPacketSize*	*1*	*0x40*	*Max packet size for endpoint zero (64).*
8	*idVendor*	*2*	*0x173A*	*Vendor ID (VID),* ***Roche.***
10	*idProduct*	*2*	*0x83A*	*Product ID (PID).*
12	*bcdDevice*	*2*	*0x0101*	*Device release number. Roche USB-****HID Protocol Version MM.mm*** *where MM is the major version and mm is the minor version in BCD format, e.g. 0x0100 represents version 1.00.*
14	*iManufacturer*	*1*	*0x01*	*Index string descriptor describing the manufacturer.*
15	*iProduct*	*1*	*0x02*	*Index string descriptor describing the Product.*
16	*iSerialNumber*	*1*	*0x03*	*Index string descriptor describing the device's serial number.*
17	*bNumConfigurations*	*1*	*0x01*	*Number of possible configurations.*

Table 2 ACUC Device Descriptor

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*blength*	*1*	*0x12*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x01*	*This descriptor type,* ***Device****.*
2	*bcdUSB*	*2*	*0x0110*	*USB specification version number,* ***Version 1.1****.*
4	*bDeviceClass*	*1*	*0x00*	*Class code.*
5	*bDeviceSubClass*	*1*	*0x00*	*Subclass code.*
6	*bDeviceProtocol*	*1*	*0x00*	*Protocol code.*
7	*bMaxPacketSize*	*1*	*0x40*	*Max packet size for endpoint zero (64).*
8	*idVendor*	*2*	*0x173A*	*Vendor ID (VID),* ***Roche.***
10	*idProduct*	*2*	*0x2106*	*Product ID (PID).*
12	*bcdDevice*	*2*	*0x0101*	*Device release number. Roche USB-****HID Protocol Version MM.mm*** *where MM is the major version and mm is the minor version in BCD format, e.g. 0x0100 represents version 1.00.*
14	*iManufacturer*	*1*	*0x01*	*Index string descriptor describing the manufacturer.*
15	*iProduct*	*1*	*0x02*	*Index string descriptor describing the Product.*
16	*iSerialNumber*	*1*	*0x00*	*Index string descriptor describing the device's serial number. Note: ACUC has no serial number*
17	*bNumConfigurations*	*1*	*0x01*	*Number of possible configurations.*

Note that the class and subclass fields are null. The class (HID) is defined in the appropriate interface descriptor. The device release number "bcdDevice" is used to identify releases of the USB-HID protocol.

5.1.2 Configuration Descriptor

Table 3 Configuration Descriptor

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x09*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x02*	*This descriptor type **Configuration**.*
2	*wTotalLength*	*2*	*0x0**22***	*Size of all data returned for this configuration. [This value must be documented prior to testing]*
4	*bNumInterfaces*	*1*	*0x01*	*Number of interfaces supported by the configuration (1 (for HID) plus the number of other interfaces supported by the device).*
5	*bConfigurationValue*	*1*	*0x01*	*Value used in Set_Configuration.*
6	*iConfiguration*	*1*	*0x00*	*Index of string descriptor for configuration.*
7	*bmAttributes*	*1*	*0x80*	*Bus powered. No remote wakeup.*
8	*MaxPower*	*1*	*0x32*	*Maximum current draw of device, in 2mA units*

wTotalLength will be computed and inserted by the base firmware. There is no need for a configuration string descriptor since only one configuration exists. Therefore, iConfiguration is left as a null value.

5.1.3 Interface and Endpoint Descriptors

Table 4 HID Interface Descriptor

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x09*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x04*	*This descriptor type, **Interface**.*
2	*bInterfaceNumber*	*1*	*0x01* *0x00*	*Number identifying this interface with:* • *0x00 for ACUC* • *0x01 for Smart Pix*
3	*bAlternateSetting*	*1*	*0x00*	*Value used to select an alternate setting.*
4	*bNumEndpoints*	*1*	*0x01*	*Number of endpoints used by this interface.*
5	*bInterfaceClass*	*1*	*0x03*	*Class code, **HID**.*
6	*bInterfaceSubclass*	*1*	*0x00*	*SubClass code, **None**.*
7	*bInterfaceProtocol*	*1*	*0x00*	*Protocol code. **No protocol**.*
8	*iInterface*	*1*	*0x00*	*Index of string descriptor for the interface. **No string**.*

Table 5 HID Descriptor

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x09*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x21*	*This descriptor type, **HID**.*
2	*bcdHID*	*2*	*0x0100*	*HID specification version compliance, **1.0**.*
4	*bCountryCode*	*1*	*0x00*	*Country code.*
5	*bNumDescriptors*	*1*	*0x01*	*Number of class descriptors.*
6	*bDescriptorType*	*1*	*0x22*	*Class descriptor, **REPORT DESCRIPTOR**.*
7	*wDescriptorLength*	*2*	*0x**032***	*Total size of class descriptor. [This value must be documented prior to testing]*

Table 6 HID Endpoint Descriptor

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x07*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x05*	*Descriptor type, **Endpoint**.*
2	*bEndpointAddress*	*1*	*0x83*	*Endpoint address and direction of transfer, **IN and endpoint #3**.*
3	*bmAttributes*	*1*	*0x03*	*Transfer type, **INTERRUPT**.*
4	*wMaxPacketSize*	*2*	*0x0040*	*The maximum packet size for this endpoint (64).*
6	*bInterval*	*1*	*0x01*	*Polling interval in milliseconds.*

5.2 String descriptors

5.2.1 Language ID String Descriptor

Table 7 Language ID String (Index 0)

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x04*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x03*	*This Descriptor Type **String**.*
2	*bString*	*2*	*0x0409*	*English Language ID.*

The index used to retrieve the language ID string descriptor via the "Get_Descriptor" standard request always defaults to zero. This descriptor is required if there are any other string indexes defined in the product in order for the strings to be interpreted properly by the USB host.

5.2.2 Manufacturer ID, Product ID, Serial Number String Descriptors

Manufacturer ID strings, Product ID strings, and Serial number strings are encoded using the UNICODE format (each ASCII character is followed by a NULL).
Note that special handling of the Serial number string may be required in the product manufacture and reporting of the string to ensure each device has a unique number.

Table 8 Manufacturer ID String (Index 1)

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x1E*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x03*	*This Descriptor Type **String**.*
2	*bString*	*xx*	*String*	*"Roche Diagnostics GmbH"*

Table 9 Smart Pix Product ID String (Index 2)

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x1C*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x03*	*This Descriptor Type **String**.*
2	*bString*	*xx*	*String*	*"Accu-Chek Smart Pix"*

Table 10 ACUC Product ID String (Index 2)

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x1C*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x03*	*This Descriptor Type **String**.*
2	*bString*	*xx*	*String*	*"Accu-Chek USB Cable"*

Table 11 Serial Number String (Index 3)

Offset (Bytes)	*Field*	*Size (Bytes)*	*Value*	*Description*
0	*bLength*	*1*	*0x1C*	*Length of this descriptor.*
1	*bDescriptorType*	*1*	*0x03*	*This Descriptor Type **String**.*
2	*bString*	*xx*	*String*	*"UInnnnnnnn", where nnnnnnnn is an 8-digit serial number*

6. HID Report Definitions

HID reports are the predefined structures used to send and receive data over the USB. Table 12 defines the HID report structures used by the host to control the operation of the device and to retrieve the status of the device. Output or OUT reports are sent (from host to device) over the control pipe using the HID class-specific "Set_Report" request. OUT reports are 64 bytes long. Input or IN reports are used to send information to the host in response to the HID commands as well as other information that is volunteered by the device. The length of the IN report is also 64 bytes. The IN and OUT report sizes were chosen to optimize data transfers while keeping the need for both large and small data packages in mind. Too large of a report size is inefficient for typically small reports. Similarly, too small a report size means larger parsing efforts for very large data packages.
Table 12 values will be listed using a byte notation due to the complexity of interpreting what should be little endian (e.g. 0x06 0x00 0xFF is actually representing a value type of 0x06 followed by the word 0xFF00 in little endian format and that the bytes are stored in location n, n+1, n+2 respectively).

Table 12 HID Report Descriptor

Item	*Type*	*Value*	*Description*
Usage Page	*Global*	*0x06 0x00 0xFF*	***Consumer Devices****
Usage	*Local*	*0x09 0x01*	***Consumer Control***
Collection	*Main*	*0xA1 0x01*	***Application***
			Custom HID Input Report
Collection	*Main*	*0xA1 0x02*	***Logical***
Usage	*Local*	*0x09 0x01*	***Vendor Usage 1**.*
Logical Minimum	*Global*	*0x15 0x00*	***0***
Logical Maximum	*Global*	*0x26 0xFF 0x00*	***255***
Report Size	*Global*	*0x75 0x08*	***8***
Report Count	*Global*	*0x95 0x40*	***64***
Input	*Main*	*0x81 0x02*	***Data, Var, Abs***
End Collection	*Main*	*0xC0*	
			Custom HID Output Report
Collection	*Main*	*0xA1 0x02*	***Logical***
Usage	*Local*	*0x09 0x01*	***Vendor Usage 1**.*
Logical Minimum	*Global*	*0x15 0x00*	***0***
Logical Maximum	*Global*	*0x26 0xFF 0x00*	***255***
Report Size	*Global*	*0x75 0x08*	***8***
Report Count	*Global*	*0x95 0x40*	***64***
Output	*Main*	*0x91 0x02*	***Data, Var, Abs***
End Collection	*Main*	*0xC0*	
End Collection	*Main*	*0xC0*	

* *A usage page of 0x06 is used instead of 0x050C to allow programs exclusive (not shared) access to the USB device.*

6.1 HID Class Reports (Commands and Responses)

HID reports are detailed in the tables below. Unless otherwise specified data counters (e.g. *blockCount*) are referenced as 1 based values while the data blocks themselves are referenced as 0 based arrays. HID reports are always represented as hexadecimal values.

OUT reports (commands in Table 13) contain two or more bytes of useful data. The first byte (BYTE 0) gives the command type (This is included for future HID expandability and is currently set to 0x01 always). The second byte (BYTE 1) gives the command value. Zero or more data bytes follow, depending on the particular command.

IN reports (responses in Table 13) are used to send various types of data from the device to the host. BYTE 0 of the HID input report gives the response type. The RUSB HID interface will only support one type of response and its identifier will be 0x01. Other values can be added if and when HID capabilities are expanded.

BYTE 1 of the HID input report gives the report value. In the case of a response to a HID command, this value is identical to the command from the host. This allows the PC to compare the response to the last command issued and flag an error if they do not match. Responses to host commands provide an acknowledgement byte and any associated data. If the device responds with a negative acknowledgment to a command, then the remaining data bytes in the response should be ignored.
Unsolicited reports can be generated at any time to log an event, error, or change in status (i.e.

these are "unsolicited" responses to the host). Currently there are no plans to use unsolicited responses for RUSB except for development and debugging purposes.
Possible values of the acknowledgement byte are as follows:

0xAA = HID_ACK:	Command successfully completed.
0xA0 = HID_NAKCMD:	Command not recognized or has incorrect parameters.
0xA2 = HID_NAKBLOCKID:	RUSB sequence block not in numerical order.
0xA5 = HID_NAKUNEXPECTED:	Required "flow" of commands not correct.
0xA6 = HID_NAKNOTREADY:	Smart Link is in MSD application, this command can actually not be handled in this mode.
0xAD = HID_NAKFWBLOCKCOUNT:	Firmware upgrade "number of blocks" failure (too few, too many).
0xAF = HID_NAKFWCHKSUM:	Firmware upgrade checksum failed.

Note that while the amount of payload bytes in each report may vary, the actual IN/OUT report size remains constant. The value 0X00 should be used as a filler character.

Unless specified otherwise, all multiple byte values in the HID commands below are LSB first (e.g. in the CmdWriteNewSeq, byte 2 is the LSB of the blockcount and byte3 is the MSB of the blockcount).

When downloading sequence data, reading sequence response data, or Firmware upgrading, the data will typically be broken into a size compatible with the report size, N (currently 64 bytes). To ensure no data is missed, a block identifier, blockID, will be included in the report. The blockID will increase sequentially with each block sent and will range from 1 to the total number of blocks. The receiver can use the blockID to verify the correct ordering of the data sent and will include the blockID in the response.

Hid commands will always operate in a synchronous manner, command followed by response. If another command is sent before the previous response is returned, that command will be ignored. The only exception to the synchronous rule is CmdReset. CmdReset may be sent like any other command, or may be used to reset the RUSB to a known state if a previous command's response has not been received.

As shown in *Figure 1*, the required flow of commands (after entering HID mode) to the Smart Pix is:

CmdWriteNewSeq,
CmdWriteBlock, …, CmdWriteBlock,
CmdRunSeq,
CmdReadDeviceData, and
CmdDataBlock, … CmdDataBlock.

Once the flow of sequence commands has begun, the order of commands must be maintained, except that CmdSwitchLed and CmdGetSmartLinkState are allowed at any point a Cmdxxxx is valid (i.e. not in between a Cmdxxxx and its Rspxxxx) and will not disrupt the required ordering. If a HID_NAKUNEXPECTED error is detected, the command flow must begin again at CmdWriteNewSeq.

CmdReset is used to place the RUSB in an IDLE (sequence not running) state and it is expected to be used only when the Cmdxxx / Rspxxxx command pattern has failed or during initial connection to the RUSB.

Default values for configuration parameters (e.g. baud rate) are only loaded on power up and not on CmdReset or when changing between MSD and HID modes (unless otherwise indicated).

Table 13 HID Output/Input Reports

Command/Response Name	***Type***	***Type (Byte 0)***	***Value (Byte 1)***	***Data (Byte 2 to Byte N-1)***
CmdWriteNewSeq	**OUT**	*0x01*	*0x10*	*Initiates loading of a new IR sequence from the PC to the RUSB. The sequence and response buffers are cleared and substitution module is initialized. Sequencer configuration is not affected.* *Bytes 2,3: blockCount, total number of blocks to be written.* *Bytes 4,5: byteCount, total size of the sequence in bytes.* *Bytes 6,7: number of steps in sequence*
RspWriteNewSeq	**IN**	*0x01*	*0x10*	*Response to CmdWriteNewSeq.* *Byte 2:* *HID_ACK: RUSB is ready to receive a new sequence.* *HID_NAKCMD: Improper command format (0 blocks, too many blocks).* *HID_NAKUNEXPECTED: A sequence is currently being output. Cannot receive a new sequence at this time.*
CmdSeqBlock	**OUT**	*0x01*	*0x11*	*One block of sequence data is sent from PC to device. RUSB uses the incrementing blockID to verify that the data blocks arrive in sequential order. Blocks are stored into the sequence memory. Must be preceded by CmdWriteNewSeq.* *Bytes 2,3: blockID, sequential identifier for data block, 1,2,…* *Bytes 4..N-1: next block of sequence data bytes.*
RspSeqBlock	**IN**	*0x01*	*0x11*	*Response to CmdSeqBlock.* *Byte 2:* *HID_ACK: Block received successfully.* *HID_NAKBLOCKCOUNT: Too many blocks received.* *HID_NAKBLOCKID: Block out of sequence, improper blockID.* *HID_NAKUNEXPECTED: A sequence is currently being output. Cannot receive a new sequence at this time.* *Byte 3: reserved* *Bytes 4,5: blockID, identifier for data block sent*
CmdRunSeq	**OUT**	*0x01*	*0x12*	*Verifies the number of sequence blocks downloaded is correct, clears the response buffer, and runs the currently stored sequence (i.e. transfer data to/from device). Must be preceded by at least one CmdSeqBlock.* *No data bytes.*

Command/Response Name	***Type***	***Type (Byte 0)***	***Value (Byte 1)***	***Data (Byte 2 to Byte N-1)***
RspRunSeq	**IN**	*0x01*	*0x12*	*Response to CmdRunSeq. Byte 2: HID_ACK: IR sequence completed (successfully or unsuccessfully). HID_NAKUNEXPECTED: A sequence is currently being output. Cannot receive a new sequence at this time. IF BYTE 2 IS ACK, THEN THE FOLLOWING BYTES ARE APPENDED: Byte 3: seqError, Error status code. OK: Sequence completed with no errors. SEQ: Unrecognized sequence command. TMO: Receive timeout occurred, see step number. CMP: Receive comparison error (see step number) UNK: Unknown error occurred, see step number. Bytes 4,5: seqSTEP, last step of sequence completed before stopping. Bytes 6,7: dataCount, size of response receive buffer in bytes.*
CmdReset	**OUT**	*0x01*	*0x13*	*Reset the RUSB and the sequence engine. Stop any sequence in progress. Reset RUSB to default settings. Host is responsible for sending additional sequences to put device into power down state. This command can be used at any time that RUSB is in HID mode to force the RUSB into a known (IDLE) state. If a sequence is running, the RUSB will look for reset indication between each Rx or Tx byte. No data bytes.*
RspReset	**IN**	*0x01*	*0x13*	*Response to CmdReset Byte 2: HID_ACK: Successful reset. HID_NAKCMD: Unsuccessful reset.*
CmdReadDeviceData	**OUT**	*0x01*	*0x14*	*Initiates reading of device data from the RUSB device. Must be preceded by CmdRunSeq. Bytes 2,3: blockCount, total number of blocks to be read Bytes 4,5: byteCount, total size of the data in bytes.*
RspReadDeviceData	**IN**	*0x01*	*0x14*	*Response to CmdReadDeviceData. Byte 2: HID_ACK: RUSB device is ready to read data. HID_NAKCMD: Improper command format (0 blocks, too many blocks). HID_NAKUNEXPECTED: A sequence is currently being output. Cannot read data at this time.*
CmdDataBlock	**OUT**	*0x01*	*0x15*	*Request for one block of sequence data to be read from device. Must be preceded by CmdReadDeviceData. RUSB uses the incrementing blockID to verify that the data blocks are read in sequential order. Must be preceded by CmdReadDeviceData. Bytes 2,3: blockID, sequential identifier for data block 1,2,…*

Command/Response Name	*Type*	*Type (Byte 0)*	*Value (Byte 1)*	*Data (Byte 2 to Byte N-1)*
RspDataBlock	**IN**	*0x01*	*0x15*	*Response to CmdDataBlock.* *Byte 2:* *HID_ACK: Block read successfully.* *HID_NAKCMD: Block not read successfully.* *HID_NAKBLOCKID: Block out of sequence, improper blockID.* *HID_NAKUNEXPECTED: A sequence is currently being output. Cannot receive sequence results at this time.* *Byte 3: reserved* *Bytes 4,5: blockID, identifier for data requested* *IF BYTE 2 IS ACK, THEN THE FOLLOWING BYTES ARE APPENDED.* *Bytes 5..N-1: Next block of response data bytes.*
CmdBeginFwUpgrade [Optional Command]	**OUT**	*0x01*	*0x40*	*Begins the USB firmware upgrade process. RUSB must be in the IDLE (sequence not running) state.* *Bytes 2..5: Number of data blocks to be transferred* *Bytes 6..9: Total number of bytes to be transferred* Note: If the device does not support this optional command a HID_NAKUNEXPECTED shall be returned in response.
RspBeginFwUpgrade [Optional Command]	**IN**	*0x01*	*0x40*	*Response to CmdBeginUsbFwUpgrade.* *Byte 2:* *HID_ACK: Successful start command.* *HID_NAKCMD: Improper command format (too many bytes, etc).* Note: If the device does not support this optional command a HID_NAKUNEXPECTED shall be returned in response.
CmdSendFwBlock [Optional Command]	**OUT**	*0x01*	*0x41*	*Sends a block of bytes (USB firmware op-codes …) to the device. Must be preceded by CmdBeginFwUpgrade.* *Bytes 2..5: blockID* *Bytes 6..N-1: Block of code bytes (zero-fill if the last block is short, 58 bytes/block if N=64).* Note: If the device does not support this optional command a HID_NAKUNEXPECTED shall be returned in response.
RspSendFwBlock [Optional Command]	**IN**	*0x01*	*0x41*	*Response to CmdSendFwBlock.* *Byte 2:* *HID_ACK: Block successfully written* *HID_NAKCMD: Failure to write block* *Byte 3: reserved* *Bytes 4..7: blockID* Note: If the device does not support this optional command a HID_NAKUNEXPECTED shall be returned in response.
CmdStartFwFlashing [Optional Command]	**OUT**	*0x01*	*0x42*	*Ends firmware download process and tells RUSB to start flashing. Must be preceeded by CmdSendFwBlock* Note: If the device does not support this optional command a HID_NAKUNEXPECTED shall be returned in response. .

Command/Response Name	***Type***	***Type (Byte 0)***	***Value (Byte 1)***	***Data (Byte 2 to Byte N-1)***
RspStartFwFlashing [Optional Command]	**IN**	*0x01*	*0x42*	*Response to CmdEndUsbFwUpgrade.* *Byte 2:* *HID_ACK: Successful start of FW upgrade.* *HID_NAKFWCHKSUM: Checksum failure.* *HID_NAKFWBLOCKCOUNT: Incorrect number of blocks received.* *Note: Flashing the firmware will typically take less than 60 seconds but may take up to 300 seconds to complete. When the update has completed, the RUSB will automatically disconnect, run the new code, and try to enumerate using the new code. After receiving the "RspStartFwFlashing, the Host PC should watch for a windows event "device removed" and "device connected" and reconnect to the RUSB. The Host PC should verify the version number to determine if the update was successful.* *The expected total size of FW and Resource may vary between 100k up to 1Meg Bytes (max. 4MB).* Note: If the device does not support this optional command a HID_NAKUNEXPECTED shall be returned in response.
CmdSwitchLED	**OUT**	*0x01*	*0x43*	*Set the LEDs on the RUSB. This command can be issued at any time a Cmdxxxxx can be issued.* *The LEDs are organized as follows:* V_{TOP} H_{LEFT} C_{LEFT} C_{RIGHT} H_{RIGHT} V_{BOT} *Byte 2: LED identifier* *0 = none are selected* *1 = horizontal LEDs (H_{LEFT} / H_{RIGHT})* *2 = vertical LEDs (V_{TOP} / V_{BOT})* *3 = horizontal and vertical LEDs)* *4 = center LEDs (C_{LEFT} / C_{RIGHT})* *5 = horizontal and center LEDs)* *6 = vertical and center LEDs* *Byte 3: flash rate (applies to LEDs selected in byte 2)* *0 = off* *1 = permanent on* *2 = flash at slow rate (~1Hz)* *3 = flash at medium rate (~2Hz)* *4 = flash at fast rate (~4Hz)* *Note:* *1. It is not possible to select All LEDs. If any combination is selected, the unused LEDs are always off* *2. In the case of the ACUC where the device only has a single LED the response to any on command shall be on and none selected shall be off.*
RspSwitchLED	**IN**	*0x01*	*0x43*	*Response to CmdSwitchLED* *Byte 2:* *HID_ACK: Command successful* *HID_NAKCMD: Command failed*

Command/Response Name	***Type***	***Type (Byte 0)***	***Value (Byte 1)***	***Data (Byte 2 to Byte N-1)***
CmdSetSmartLinkState	**OUT**	*0x01*	*0x44*	*Sets the state of the RUSB device. This command can only be issued when the RUSB is in an IDLE (sequence not running) state. If HID mode is selected, the sequence and response buffers are cleared and substitution module is initialized. The sequencer configuration is reset to defaults.* *Byte 2: Operational state* *0x00: Mass Storage Mode* *0x01: HID Mode*
RspSetSmartLinkState	**IN**	*0x01*	*0x44*	*Response to CmdSetSmartLinkState. This typically takes less than 2 seconds, but may be longer if the Smart Link is in MSD mode and is in the middle of a task.* *Byte 2:* *HID_ACK: Command successful* *HID_NAKCMD: Command failed or state not supported* *HID_NAKUNEXPECTED: The MSD mode is in the middle of a task. The command can be resent* *.* *Note: If the USB device only supports a HID interface then the response to setting the device into Mass Storage Mode from the CmdSetSmartLinkState command, shall always be HID_NAKCMD.*
CmdGetSmartLinkState	**OUT**	*0x01*	*0x45*	*Gets the state of the RUSB device. This command can be issued at any time a Cmdxxxxx can be issued.*
RspGetSmartLinkState	**IN**	*0x01*	*0x45*	*Response to CmdGetSmartLinkState* *Byte2:* *HID_ACK: Command successful* *HID_NAKCMD: Command failed or state not supported* *Byte 3: reserved* *Byte 4: Operational mode* *0x00: Mass Storage Mode* *0x01: HID Mode* *Byte 5: reserved, can be any value* *Byte 6: FW- Version "main"* *Byte 7: FW- Version "sub"* *Byte 8: FW- Version "release"* *Byte 9: FW- Version "auxiliary byte"* *Byte 10: Resource - Version "main"* *Byte 11: Resource - Version "sub"* *Byte 12: Resource - Version "release"* *Byte 13: Resource - Version "auxiliary byte"* *Bytes 14..23: 10 bytes reserved for future use* *Bytes 24..27: RUSB serial number (little endian)* *Note: The ACUC shall always return all zeros* *Bytes 28..29: maximum size of sequence buffer (little endian)* *Bytes 30..31: maximum size of response buffer (little endian)* *Bytes 32..41: 10 bytes reserved for future use* *Note:* *The FW version covers both the HID and MSD firmware.*

7. Sequence Engine API

It is desirable to create a method for sending and receiving characters between the host application and an IR device that is very flexible. The serial interactions with IR devices can involve tight timeouts at the command level and at the inter-character level; therefore it is also desirable to embed the timing critical portions of the communication protocol within RUSBand remove any critical response timing from the PC or the USB communication controlling the RUSB. To this end, a sequencing language (SL) has been developed. The SL will contain primitive serial commands that, when combined, will allow complicated interaction sequences to be sent and received from various devices. This section describes the elements of the SL and will provide an example of how to combine them to extract information from a supported Roche IR device.

7.1 Command Summary

Table 14 gives a quick summary of the *RUSB*Sequence commands. The table shows the arrangement of Sequence commands wrapped inside of the HID reports. For a detailed description of the HID commands and responses, refer to the section 6.1.

Table 14Typical HID Reports with Sequence Payload

Byte1 (Cmd)	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
CMD**WRITENEWSEQ**	blockCount		byteCount		numSeqSteps	
RSP**WRITENEWSEQ**	ackCode	-	-	-	-	-
CMD**SEQBLOCK**	blockID		seqData[]			
RSP**SEQBLOCK**	blockID		ackCode	-	-	-
CMD**RUNSEQ**	-	-	-	-	-	-
RSP**RUNSEQ**	ackCode	errCode	seqStep		dataCount	
CMD**READDEVICEDATA**	blockCount		byteCount			
RSP**READDEVICEDATA**	ackCode	-	-	-	-	-
CMD**DATABLOCK**	blockID		-	-	-	-
RSP**DATABLOCK**	blockID		ackCode	rspData[]		
CMD**RESETSEQ**	-	-	-	-	-	-
RSP**RESETSEQ**	ackCode	-	-	-	-	-

7.2 Sequence API

The sequence language is designed to form the basic building blocks needed to encapsulate the timing critical elements of IR device communication. Typically, this involves sending a short sequence of command bytes in a half-duplex, echoing mode, and then receiving a varying number of response bytes. The response is interpreted, and another command sequence is then sent until the function needed has been completed.
The sequencing language will consist of a series of data bytes that will be interpreted by *RUSB*.
A sequence may have multiple steps made from sequencing primitives in order to accomplish a data-gathering task.
A sequencing state machine will execute the sequence steps and be in control of the timing of the sequence, removing any PC variability from time-critical device interactions.
The *RUSB* will report the result of sequences via the RspReadDeviceData command and the Response buffer. At the beginning of each sequence, the Response buffer will be cleared.

The sequence buffer must be able to store at least 500 bytes (maximum 65535). This translates roughly into about 100 sequence commands (500 bytes / ~5 bytes per sequence command).
The response buffer must be able to store at least 500 bytes (maximum 65535) of device

response. This is sufficient to hold a maximum size (255 byte) ICI data packet which is used as a maximum packet benchmark.

The user should determine the sequence and response buffer sizes by requesting the sizes via CmdGetSmartLinkState.

The sections below describe the language primitives.

7.2.1 Common Format

To support an extensible command set, each command will have the following common format:

byte0	byte1	byte2	byte3	…	byteN
command	bytecnt	p1	p2	…	pN

where:

- command: the type of command
- bytecnt: the number of parameter or data bytes to follow [0..255]
- p1: optional parameter
- p2: optional parameter
- pN: optional parameter – usually data

Control Characters:

The examples that follow will use abbreviations for some of the standard ASCII control characters to improve readability.

Description	Abbreviation	Hex Value
Start of header	<soh>	0x01
Start of transmission	<stx>	0x02
End of data transfer block	<etx>	0x03
End of data transfer	<eot>	0x04
Information received or acknowledged	<ack>	0x06
Tab	<tab>	0x09
Carriage return	<cr>	0x0D
Information not received correctly or not acknowledged	<nak>	0x15
Abort data transfer, wake-up	<can>	0x18

Table 15: Control Character Abbreviations

7.2.2 Sequence Errors

During the course of sequence execution, a number of errors may occur. The error will be reported in the errCode variable. The currently defined errors are:

Name	Index	Description
SEQ_ERR_NONE	0	No error.
SEQ_ERR_SEQ	1	An unrecognized sequence command appeared.
SEQ_ERR_TMO	2	A timeout on waiting for a byte occurred.
SEQ_ERR_CMP	3	A comparison error occurred.
SEQ_ERR_RESPOVR	4	Overflow on storing RX bytes to response buffer.
SEQ_ERR_BADPARAM	5	Bad parameters found on decoding command.

SEQ_ERR_CFG	6	Bad configuration parameters found during sequence.
SEQ_ERR_TERMINATED	7	Sequence was terminated before finish because of cmdReset
SEQ_ERR_UNK	8	unknown other error.

7.2.3 Command Summary

The following table gives a quick summary of the RUSB sequence commands.

Index	Command	Count	p1		p2		p3
01	LOOPBACK	Special Format					
02	RX	5	rxCount	rxFlags	rxCompare	rxMaxL	rxMaxH
03	RXCNT	3	rxcCount		rxcFlags		rxcOffset
04	TX	n+1	txFlags		txBytes[n]		-
05	TXECHO	n+1	txeFlags		txeBytes[n]		-
06	WAIT	1	wtDelay10ms		-		-
07	CFG	n	cfgFlags		cfgIndex		cfgValue

7.2.4 Loopback(01)

The LOOPBACK command allows the PC to test the USB communication portion of the system and to speed development of application code. The application developer can simulate communication with the *RUSB* and an associated device by sending the LOOPBACK command filled with the expected (or unexpected) response data. The LOOPBACK command copies the data following the LOOPBACK index directly into the response buffer and sets the sequence error code, sequence step, response size, and response bytes.

Note: The LOOPBACK command is designed to be the only command in a sequence (because it simulates the response from a sequence) and should NOT be combined with any other commands.

Format:

Command	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7
LOOPBACK(01)	dataCount		ackCode	errCode	seqStep		data[]…

where:

• dataCount:	(simulated) number of response bytes
• ackCode:	(simulated) acknowledgement code
• errCode:	(simulated) sequence error code
• seqStep:	(simulated) sequence step number
• data[]:	(simulated) response bytes from device communication or configuration

7.2.5 Rx(02)

The RX command causes RUSB to receive from 1 to 255 bytes, receive a variable number of bytes (determined using the Rxcnt command), scan for a specific byte, or receive bytes until an auto end detection (AED) via the IR link and place them in the receive buffer.

To read a variable number of bytes, the sequence must first use the Rxcnt command to interpret the received bytes that make up the packet count. This interpreted count is saved and can be used as the receive count, pktCount, for the Rx command when the rxPKT flag is selected.

To scan for a specific byte, the rxSCAN flag is used. The RUSB will then look for rxCompare for up to rxMaxBytes bytes. If the byte is found, the sequencer continues to the next step; otherwise an error is generated.

To receive bytes until auto end detection (AED) occurs, the rxAED flag is used. The RUSB will then receive up to rxMaxBytes bytes until a byte-to-byte timeout (specified by the cfgRX_BYTE_TO_BYTE_TIMEOUT configuration value) occurs. When a timeout elapses or the maximum bytes have been received, the sequencer continues to the next step; otherwise an error is generated.

When rxCMP is set, the last byte of a fixed count, variable (rxPKT), or AED Rx command will be compared to the rxCompare value; if equal, the sequence continues, if not equal, the sequence stops.

When rxSUBST is set, the received bytes are compared to the Rx substitution pattern. If the pattern matches, the bytes are replaced by the Rx substitution. The Rx substitution history is reset whenever a new sequence is downloaded or when the previous step was not a Rx command with substitution enabled.

The Rx command places any received bytes (processed bytes if SUBST is enabled) into the response buffer. It is possible for Rx bytes to be received that are not handled by an Rx or Rxcnt step. If an Rx or Rxcnt step is preceded by a Tx or TxEcho step, any unhandled receive bytes will be cleared.

The rxSCAN, rxAED, and rxPKT functions are exclusive and only one (or none) of these functions can be used in an Rx step. If more than one bit is selected, the priority will be rxPKT, rxAED, then rxSCAN and the other bits will be ignored.

Format:

Command	Bytecnt	p1	p2	p3	p4	p5
RX	5	rxCount	rxFlags	rxCompare	rxMaxBytesL	rxMaxBytesH

where:

- rxCount: number of characters to receive, 0 for packet read
- rxFlags: flags to control the receive operation
  - rxCMP(bit0): 1 = last receive byte must match rxCompare or sequence will stop. Ignored if rxSCAN = 1.
  - rxSCAN(bit1): 1 = scan input bytes for rxCompare for up to rxMaxBytes bytes. rxCount = 0.
  - rxAED(bit2): 1 = receive input bytes until a byte-to-byte timeout (RX_BYTE_TO_BYTE_TIMEOUT) elapses for up to rxMaxBytes bytes. rxCount = 0. If rxCMP = 1, compare last byte received.
  - rxPKT(bit3): 1 = use pktCount for number of receive bytes (see Section 7.2.6 for an explanation of how pktCount is created). rxCount = 0. If rxCMP = 1, compare last byte received.
  - rxSUBST(bit4): 1 = enable substitution string replacement, 0 = no substitution
  - bits 5..7: reserved, set to 0
- rxCompare: if comparing (rxCMP = 1), comparison byte, otherwise 0

- rxMaxBytes: maximum number of characters (must be >=1) to receive if rxAED or rxSCAN = 1, 0 otherwise

Example: Receive Any 1 Byte

Command	Bytecnt	p1	p2	p3	p4	p5
RX(02)	5	1	0	0	0	0

Example: Receive <stx>

Command	Bytecnt	p1	p2	p3	p4	p5
RX(02)	5	1	1	<stx>	0	0

Example: Receive 7-bytes ending in <eot>

Command	Bytecnt	p1	p2	p3	p4	p5
RX(02)	5	7	1	<eot>	0	0

Example: Receive pktCount *bytes*

Command	Bytecnt	p1	p2	p3	p4	p5
RXCNT(03)	5	2	1	0	-	-
RX(02)	3	0	2	0	0	0

If the received bytes are:

30 33 41 42 43

The Rxcnt command will read a 2 byte ASCII Hex packet count (30 33) and interpret as a count of 3 and save the value in pktCount. The Rx command will then read pktCount bytes (rxPKT flag is set) and "ABC" will be read.

Example: Find the next tab, <tab>, for up to 20 characters

Command	Bytecnt	p1	p2	p3	p4	p5
RX(02)	5	0	2	<tab>	20	0

If the received bytes are:

30 33 35 31 <tab> 41 42 43

The Rxcnt scan will read 30 33 35 31 and <tab> into the response buffer, then continue to the next sequence step. (If the next step is a transmit, the 41 42 43 will not be placed in the response buffer.)

Example: Receive up to 300 Auto End Detection bytes ending in <etx>

Command	Bytecnt	p1	p2	p3	p4	p5
RX(02)	5	0	5	<etx>	0x2C	0x01

7.2.6 Rxcnt(03)

The RXCNT command causes RUSB to receive and interpret one or more bytes into a packet count, pktCount, to allow a variable number of bytes to be read. An optional offset is then added to the interpreted count, the count is stored in pktCount, and the count can be used in a subsequent RX command (by setting the rxPKT flag). The RXCNT command can interpret single or multi-byte binary, ASCII decimal, and ASCII hex fixed length values (i.e. the size of the packet count in bytes must be known). The converted count value cannot be larger than 65535. Leading spaces (0x20) in ASCII hex or decimal conversion will be treated as zeroes.

The pktCount will be initialized to 0. Once a pktCount is interpreted, that pktCount will remain until another pktCount replaces its value.
The Rxcnt command places any received bytes (processed bytes if SUBST is enabled) into the

response buffer. It is possible for Rx bytes to be received that are not handled by an Rx or Rxcnt step. If an Rx or Rxcnt step is preceded by a Tx or TxEcho step, any unhandled receive bytes will be cleared.

Format:

Command	Bytecnt	p1	p2	p3
RXCNT	3	rxcCount	rxcFlags	rxcOffset

where:

- rxcCount: number of characters to receive. Maximum value is 2 if rxcBIN is selected, 4 it the rxcAHEX) is selected, and 5 if (rxcADEC) is selected.
- rxcFlags: flags to control the receive operation
    - (bits0..2) Type of packet count
      rxcBIN: value=0, binary count value
      rxcAHEX: value = 1, ASCII hex count value
      rxcADEC: value= 2, ASCII decimal count value
      values 3-7, reserved
    - rxcLSB(bit3) 1 = least significant count byte first (binary type only)
    - rxcSUBST(bit4) 1 = enable substitution string replacement, 0 = no substitution
    - bits 5..7: reserved, set to 0
- rxcOffset signed offset to add to count decoded

Example: Receive 2-byte ASCII Hex Packet Count with offset +3

Command	Bytecnt	p1	p2	p3
RXCNT(03)	3	2	1	3

Example: Receive 3-byte Binary, LSB first Packet Count with no offset

Command	Bytecnt	p1	p2	p3
RXCNT(03)	3	3	8	0

7.2.7 Tx(04)

The TX command causes one character to be output via the IR link. If RUSB was previously receiving bytes, a transmit delay (at least 10ms for the ICI protocol) will be inserted before sending the character(s) to allow the far end receiver to switch from transmit to receive mode.

Format:

Command	Bytecnt	p1	p2	p3
TX	n+1	txFlags	txBytes[n]	

where:

- txFlags
    - txSUBST(bit0) 1 = enable substitution string replacement, 0 = no substitution
    - bits 1..7: reserved, set to 0
- txBytes[n] the n bytes to transmit

Example: Transmit 5 bytes

Command	Bytecnt	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6
TX(04)	6	0	‘H’	‘e’	‘l’	‘l’	‘o’

Example: Transmit 5 Bytes with Substitution

Command	Bytecnt	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6
TX(04)	6	1	0x7F	0x7F	'B'	'y'	'e'

If the Substitution pattern is set for 7F 7F ⇔ 7F 7F 01, the actual data output by this command would be:

7F 7F **01** 42 79 65

7.2.8 Txecho(05)

The TXECHO command causes a string of bytes to be output via the IR link in a half-duplex, wait for echo manner. If RUSB is in the receive mode, a transmit delay will be inserted before sending each byte to allow the receiver to switch from transmit to receive mode.

The Txcnt command places any echoed bytes into the response buffer. Substitution will not be available using the Txcnt command due to the nature of the command. If substitution is required, use a combination of the Rx and Tx commands.

Format:

Command	Bytecnt	p1	p2	p3
TXECHO	n+1	txeFlags	txeBytes[n]	-

where:

- txeFlags
  - txeLAST(bit0) 1 = don't wait for echo on last byte
  - bits 2..7: reserved, set to 0
- txeBytes[n] the n bytes to transmit

Example: Transmit the Get Model Number Command (ICI command)

Transmit the string "C<tab>4<cr>", waiting for an echo on all but the last character (<cr>).

Command	Bytecnt	p1	p2	p3
TXECHO(05)	7	1	C<tab>4<cr>	

7.2.9 Wait(06)

The WAIT command provides a delay method for the sequence. If delays greater than the maximum delay (2.55sec) are needed, additional wait commands can be used. Because the 10ms timer is not synchronized to the sequence execution, the delay value will give a range of time, e.g. a delay of 20 will give a delay of 190-200ms. With a one byte delay, a range from 0ms to 2.55s is possible.

Format:

Command	Bytecnt	p1	p2	p3
WAIT	1	wtDelay10ms	-	-

where:

- wtDelay10ms the delay in 10ms ticks, range [0..255], delay time = value * 10ms

7.2.10 Cfg(07)

The ConFiGure command allows the PC to configure parameters within the RUSB. The CFG command acts like a sequence step. If there is an error in configuration, it will be reported in the

sequence response message with an error code and the sequence step on which the error occurred.
When cfgSET is zero, the configuration value(s) for the index requested will be placed in the response buffer and will be returned in the sequence response. No cfgValue array items are passed in the command and n = 0. The PC is responsible for interpreting the response bytes correctly. (see format below)
When cfgSet is one, the configuration values (cfgValue{0], cfgValue[1] … cfgValue[n]) are included in the command beginning at parameter 3 (p3, p4 … p(n+3) respectively) and n = number of parameters corresponding to the configuration index. (see format below)

7.2.10.1 Byte-to-Byte Timeout

The byte-to-byte timeout specified by the cfgRX_BYTE_TO_BYTE_TIMEOUT is active for all Rx step operations. The first byte timeout for each RX step is equal to the RX timeout. All following bytes will use the cfgRX_byte_to_byte _timeout value.

7.2.10.2 Substitution

The Substitution configuration allows RUSB to handle protocols that require byte stuffing and removal (e.g. the pump protocols). The substitution configuration allows a pattern and a substitution to be setup for both Rx and Tx. Basically, when the receive or transmit pattern matches the corresponding substitution pattern, RUSB will "process" the bytes and make the appropriate substitution before placing the bytes in the response buffer or outputting the bytes.

Rx Rules: The Rx substitution will look back into a history of received characters in order to determine a substitution match. The substitution history is cleared when:

- a new sequence is downloaded
- the previous command was not an Rx or RxCnt command with substitution enabled

Tx Rules: The Tx substitution must operate on a look-ahead basis in order to make a substitution before any substituted characters are output. Tx substitution is initialized:

- on every Tx command

Sequence generators must carefully construct sequences that might combine successive Tx commands to ensure that a substitution pattern does not span two consecutive Tx commands. Sequence generation tools should re-factor consecutive Tx commands into one Tx command to eliminate this problem.
See section 7.3.2.1 and 7.3.2.2 for examples of substitution.

Format:

Command	Bytecnt	p1	p2	p3…
CFG	n+2	cfgFlags	cfgIndex	cfgValue[n]

where:

- cfgFlags flags to control the operation of the configuration command
  - cfgSET(bit0) 0 = get the indexed parameter, 1 = set the indexed parameter
- cfgIndex The index of the parameter. Defined indexes are listed below
  - cfgSERIAL_IR Index = 0. Set parameters for serial IR communication.
    cfgValue[0]: Baud rate for IR communication, 0 = 2400, 1 = 4800, 2 = 9600, 3 = 19200, 4 = 38400, 5 = 57600, 6 = 115200. Default = 2 (9600 baud)
    cfgValue[1]: Number of bits: 7 = 7bits, 8 = 8 bits, all others reserved.

cfgValue[2]:	Parity: 0 = none, 1 = odd, 2 = even, all others reserved.
cfgValue[3]:	Number of stop bits: 1 = 1bit, 2 = 2 bits, all others reserved.

- cfgRX_TO_TX_TIMEOUT Index = 1. Time delay when transitioning from receiving bytes to transmitting bytes in 2ms ticks; range [0..255], 0 indicates no timeout. Time delay = value * 2ms. Default = 6 (10-12ms). Note: when a sequence starts with a Tx step, the system will first wait the RxToTxTimeout before outputting the first Tx byte.
  cfgValue[0]: delay value.

- cfgRX_TIMEOUT Index = 2. Time allowed when waiting for the next receive byte in 20ms ticks, range [0..255], 0 indicates no timeout. Timeout = value * 20ms. Default = 300 (3.0 sec)
  cfgValue[0]: delay value.

- cfgTX_PATTERN Index = 3. Pattern of up to 8 bytes to look for when transmitting.
  cfgValue[0]: Length of pattern.
  cfgValue[1..8]: Pattern bytes

- cfgTX_SUBST Index = 4. Pattern of up to 8 bytes to substitute when transmitting
  cfgValue[0]: Length of substitution.
  cfgValue[1..8]: Substitution bytes

- cfgRX_PATTERN Index = 5. Pattern of up to 8 bytes to look for when receiving.
  cfgValue[0]: Length of pattern.
  cfgValue[1..8]: Pattern bytes

- cfgRX_SUBST Index = 6. Pattern of up to 8 bytes to substitute when receiving
  cfgValue[0]: Length of substitution.
  cfgValue[1..8]: Substitution bytes

- cfgRX_BYTE_TO_BYTE_TIMEOUT Index = 7. Maximum time delay between received bytes when in the receive mode. Time delay is in 2ms ticks; range [0..255], 0 indicates no timeout. Time delay = value * 2ms. Default = 50 (100-102ms).
  cfgValue[0]: value for timeout.

- cfgTX_BYTE_TO_BYTE_WAIT Index = 8. Time delay between transmitted bytes when in the transmit mode. Time delay is in 1ms ticks; range [0..255], 0 indicates no delay. Time delay = value * 1ms. Default = 0 (no wait cycles).
  cfgValue[0]: delay value.

Example: Set serial baud rate to 9600, 8 bits, even parity, 1 stop bit

Command	Bytecnt	p1	p2	p3	p4	p5	p6
CFG (07)	6	1	0	2	8	0	1

Example: Get Rx-to-Tx Timeout

Command	Bytecnt	p1	p2	p3
CFG (07)	3	2	1	0

Rx-to-Tx timeout value is placed in the next available response buffer position and can be read with CMDREADDEVICEDATA.

Example: Setting up substitution for Spirit pump communication

Command	Bytecnt	p1	p2	p3	p4..
CFG (07)	5	2	3	2	7F 7F
CFG (07)	6	2	4	3	7F 7F 01
CFG (07)	6	2	5	2	7F 7F 01
CFG (07)	5	2	6	3	7F 7F

The Spirit pump communication requires a low-level byte substitution when transmitting and receiving to allow accurate preamble detection (0x7F 0x7F 0x7F). When transmitting (other than the preamble data), whenever 0x7F 0x7F is found in the transmit stream, the pattern 0x7F 0x7F 0x01 must be output. Whenever 0x7F 0x7F 0x01 is found in the receive data stream, it must be replaced with 0x7F 0x7F. The SUBST configuration parameters allow the sequence developer to setup transmit and receive substitution strings that can be used when transmitting or receiving via a flag option in the sequencing commands.

The commands above set up the substitution pattern for the Spirit pump. (Note: the xxSUBST flag must be used on any transmit or receive sequence steps when substitution is desired. See Section 7.3.2.1 for an example.)

7.3 Examples

To demonstrate how the sequences can be used to extract information from devices, the following examples have been prepared. An asterisk (*) beside a step number indicates the sequence may terminate (in error) on the step.

In the following examples, bytes 5 and 6 of an Rx are assumed to be set to 0 unless otherwise specified.

7.3.1 ICI-compliant Devices

7.3.1.1 Wake-Up

To wake up an ICI device, the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TX	2	0	<soh>		Wake up
2	WAIT	1	5			Wait 50-60ms
3	TX	2	0	<soh>		Wake up
4	WAIT	1	5			Wait 50-60ms
5	TX	2	0	<can>	-	Wake up device
6*	RX	5	1	0	0	Accept anything (<nak>)
7	TX	2	0	<cr>	-	Finish wake up

8*	RX	5	1	0	0	Accept anything (<nak>)
						Device is now awake

7.3.1.2 Read and Clear Status

To prepare an ICI device for communication and reset any error, the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1*	TXECHO	3	txeLAST	<0B><cr>		Echo read and clear cmd
2*	RX	5	1	rxCMP	<ack>	Accept <ack> only
3*	RX	5	1	rxCMP	<stx>	Accept <stx> only
4*	RXCNT	3	2	rxcAHEX	0	Get 2-byte ASCII hex packet count, offset 0
5*	RX	5	0	rxPKT	0	Read packet
6*	RX	5	2	0	0	Read CRC
7*	RX	5	1	rxCMP	<eot>	Read <eot> only
8	TX	2	0	<ack>	-	Acknowledge packet
9*	RX	5	1	rxCMP	<ack>	Accept <ack> only
						Device is ready to communicate

7.3.1.3 Read Number of Records

To read the number of records in the device, the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TXECHO	3	txeLAST	<60><cr>		Echo number of records command
2*	RX	5	1	rxCMP	<ack>	Accept <ack> only
3*	RX	5	1	rxCMP	<stx>	Accept <stx> only
4*	RXCNT	3	2	rxcAHEX	0	Get 2-byte ASCII hex packet count
5*	RX	5	0	rxPKT	0	Read packet
6*	RX	5	2	0	0	Read CRC
7*	RX	5	1	rxCMP	<eot>	Read <eot> only
8	TX	2	0	<ack>	-	Acknowledge packet
9*	RX	5	1	rxCMP	<ack>	Accept <ack> only
						Packet has number of records

7.3.1.4 Read First Record

To read the first record in the device, the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TXECHO	10	txeLAST	<61><tab>1 <tab>NNN <cr>		Echo read records where NNNN is the 1 to 4 digit number of records in the device
2*	RX	5	1	rxCMP	<ack>	Accept <ack> only
3*	RX	5	1	rxCMP	<stx>	Accept <stx> only
4*	RXCNT	3	2	rxcAHEX	0	Get 2-byte ASCII hex packet count
5*	RX	5	0	rxPKT	0	Read packet
6*	RX	5	2	0	0	Read CRC
7*	RX	5	1	0	0	Accept <etx> or <eot>
						Packet has first record's data

7.3.1.5 Read Next Record

To read the next record in the device after starting with the read first record sequence, the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TX	2	0	<ack>	-	Request next record
2*	RX	5	1	rxCMP	<stx>	Accept <stx> only
3*	RXCNT	3	2	rxcAHEX	0	Get 2-byte ASCII hex packet count
4*	RX	5	0	rxPKT	0	Read packet
5*	RX	5	2	0	0	Read CRC
6*	RX	5	1	0	0	Accept <etx> or <eot>
						Packet has next record's data

To finish reading records the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TX	2	0	<ack>	-	Finish reading
2*	RX	5	1	rxCMP	<ack>	Accept <ack>
						Ready for commands

To cancel reading records the following sequence should be sent:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TX	2	0	<can>	-	Cancel record read
2*	RX	5	1	0	0	Accept <ack> or <nak>
						Reading is cancelled. PC must issue read and clear

7.3.1.6 Power Down

To complete the device interaction and power down the device to leave it in a low power state, use the following sequence:

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TXECHO	3	txeLAST	<1d><cr>		Power down device
2*	RX	5	1	rxCMP	<ack>	Accept <ack>
3*	RX	5	1	rxCMP	<ack>	Accept <ack>
						Device is powered down if <ack> response

7.3.2 Spirit Pump

7.3.2.1 Send a Packet

To work with the Spirit pump, it is assumed that the proper substitution patterns have already been entered into the RUSB configuration (see Section 7.2.10 examples). The following packet sends the data "ABCD" to a Spirit.

step	byte0	byte1	byte2	byte3	byte4+	Description
1	TX	4	0	7F 7F 7F		preamble
2	TX	4	txSUBST	12 50 04		DLL version 12 Frame type 50 Data length 04
3	TX	5	txSUBST	01 02 03 04		Destination 01 02 03 04
4	TX	5	txSUBST	89 7F 7F EF		Source 89 7F 7F EF
5	TX	5	txSUBST	41 42 43 44		Data "ABCD"
6	TX	3	txSUBST	CL CH		CRC word (2 bytes)
7	TX	2	0	01		Protocol recommends ending with 01
						Packet has been sent

This sequence will result in the following bytes being output to the IR transmitter:

```
7F 7F 7F  12  50  04  01 02 03 04  89 7F 7F 01 EF  41 42 43 44  CL CH  01
```

Note the automatic insertion of 01 in the data due to the substitution pattern match.

7.3.2.2 Receive a Packet

To work with the Spirit pump, it is assumed that the proper substitution pattern has already been entered into the RUSB configuration (see Section 7.2.10 examples). The following packet receives the data "SPIRIT" from a Spirit.

step	byte0	byte1	byte2	byte3	byte4+	Description
1	RX	5	3	0	0	preamble
2	RX	5	2	rxSUBST	0	DLL version 12 Frame type 50
3	RXCNT	3	1	rxcSUBST | rxcBIN	8	Binary data length, 06, add 8 to compensate for destination and source addresses
4	RX	5	0	rxSUBST | rxPKT	0	Destination 01 7F 7F 04 Source 89 AB CD EF Data “SPIRIT”
5	RX	5	2	rxSUBST	0	CRC word (2 bytes)
						Packet has been received

This sequence will read the following data bytes properly:

7F 7F 7F 12 50 06 01 7F 7F 01 04 89 AB CD EF 53 50 49 52 49 54 CL CH

Note the automatic removal of 01 in the data due to the substitution pattern match.

7.3.2.3 Receive a Packet using Auto End Detection

To work with the Spirit pump, it is assumed that the proper substitution pattern has already been entered into the RUSB configuration (see Section 7.2.10 examples). The following packet receives the data “SPIRIT” from a Spirit.

step	byte0	byte1	byte2	byte3	byte4	byte5	byte6	Description
1	RX	5	0	4	0	0	2	Receive preamble, DLL version, frame type, length, destination, source, data, and CRC (up to 512 bytes)
								Byte-to-byte time elapses after last CRC byte
								Packet has been received

This sequence will read the following data bytes properly:

7F 7F 7F 12 50 06 01 7F 7F 01 04 89 AB CD EF 53 50 49 52 49 54 CL CH

Note the automatic removal of 01 in the data due to the substitution pattern match.