Logitech HID++2.0 Draft Specification

04 June, 2012

Contents

Disclaimer

THIS SPECIFICATION IS LICENSED AND PROVIDED BY LOGITECH "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LOGITECH OR ANY OF ITS AFFILIATED COMPANIES BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SPECIFICATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Introduction

Logitech's HID++2.0 is the protocol over HID used to access most of Logitech HID devices.

This document describes the basics of HID ++ 2.0 protocol, including versioning, the structure of functions and the following features: entry point, retrieving firmware information such as version and build numbers and device name and type (such as keyboard, mouse, etc.). An example device transaction concludes this document.

Protocol Versioning

HID++ protocol should have a versioning support to allow forward compatibility and versioning API should be obviously independent of the various protocols.
Since version 1.0 of the protocol does not have explicit versioning support, we need a way to discriminate version 1.0 from 2.0 and further.

Version 1.0 of the protocol does not implement RegisterAccessID 0x00. Sending a request with this value generates an error message, One solution is to use this particularity to implement a GetProtocolVersion function.

version = GetProtocolVersion()

Request: 0x10.DeviceIndex.**0x00.0x1n**.0x00.0x00.**0xUU**

n in 0x1n is SwID

0xUU is "ping data" defined by SW

Response:

HID++ 1.0: 0x10.DeviceIndex.**0x8F**.0x00.0xFn.0x01.0x00 (ERR_INVALID_SUBID response)

HID++ 2.0: 0x10.DeviceIndex.**0x00.0x1n**,0x02.0x00.**0xUU**

HID++ XX.YY: 0x10.DeviceIndex.**0x00.0x1n**,0xXX.0xYY.**0xUU**

Ping same as Protocol versioning

response = Ping(0xXX)

Request: 0x10.DeviceIndex.**0x00.0x1n**.0x00.0x00.**0xUU**

Response:

HID++ 2.0: 0x10.DeviceIndex.**0x00.0x1n**,0x02.0x00.**0xUU**

This makes the ping request appear as a hidden 0xE function of the IRoot feature so that firmware and software can use the normal feature/function dispatch mechanism to handle it.
to simplify implementation in both FW and SW is to make the ping request "look" like a normal feature request by putting the SwId in its normal place. The "n" in 0xEn below would be the SwId. The change in the high nibble from F to E is to avoid conflict with the GetProtocolVersion request.

Protocol HID++2.0 essential features

Functions

A function is composed of a request sent by the host PC followed by one or more responses returned by the HID++ 2.0 device/receiver. Within a given feature, each function is defined by a function identifier (assigned from a pool of 16 identifiers shared with ASEs, see below). Requests can have up to 3 or 16 bytes of parameters, while responses return the same quantity of results. Most functions will be read or write functions (although functions which do both or neither are also allowed).

Unless otherwise specified, the protocol is big-endian, meaning that a 16bits value sent on bytes 4 and 5, will have its MSB in byte 4 and its LSB in byte 5.

Requests

0	1	2	3		4	5	6
0x10	Dev. index [1 .. 255]	Feature index [1 .. 254]	Fctn/ASE id. [0 .. 15]	Sw. id. [1 .. 15]	Parameters		

0	1	2	3		4	5	6	...	19
0x11	Dev. index [1 .. 255]	Feature index [1 .. 254]	Fctn/ASE id. [0 .. 15]	Sw. id. [1 .. 15]	Parameters				

Responses

0	1	2	3		4	5	6	...	19
0x11	Dev. index [1 .. 255]	Feature index [1 .. 254]	Fctn/ASE id. [0 .. 15]	Sw. id. [1 .. 15]	Results				

0	1	2	3		4	5	6	...	19
0x11	Dev. index [1 .. 255]	Feature index [1 .. 254]	Fctn/ASE id. [0 .. 15]	Sw. id. [1 .. 15]	Results				

The device index, feature index, function identifier, and software identifier are always returned unchanged.

All functions must respect the following rule:

All parameters in a request must always be repeated in the response:

Any parameter that is fully supported must be repeated "as is".

Any parameter that is only "partially supported", must be returned as supported.

The following examples should help to better understand these requirement:

A read command where an address (i.e., memory address, register address, etc.) is given as parameter must return the address and the data in the response (and not just the address).

A write command which sets a collection of bits or bit fields, should return the same value where all unsupported bits and bit fields have been masked to their default values (usually 0).

A command such as open-lock, erase-memory, etc. should return its parameters unchanged.

Note that there is no requirement to implement partial support. Each feature designer is free to decide if "partially correct" parameters should return and error or be "partially supported." However, if partial support is implemented, then the parameters must be returned as supported.

Application-specific events (ASEs)

An application-specific event (ASE) is a notification sent by an HID++ 2.0 device/receiver, which is linked to a previously-received function and primarily destined to the software that sent this command. In this case, the software identifier associated with the function is repeated in the ASE. Within a given feature, each ASE is defined by an ASE identifier (assigned from a pool of 16 identifiers shared with functions, see above). ASEs have up to 3 or 16 bytes of data.

Broadcast events

A broadcast event is an notification sent by an HID++ 2.0 device/receiver, which is not destined to any particular software (note that it might or might not be linked to a previously-received function). Within a given feature, each broadcast event is defined by an broadcast event identifier. Broadcast events have up to 3 or 16 bytes of data.

Software identifier (4 bits, unsigned)

A number uniquely defining the software that sends a request. The firmware must copy the software identifier in the response but does not use it in any other ways.

0 Do not use (allows to distinguish a notification from a response).

HID++2.0 error codes

Error	Value	Comment
NoError	0	Not an error
Unknown	1	
InvalidArgument	2	
OutOfRange	3	
HWError	4	
Logitech internal	5	
INVALID_FEATURE_INDEX	6	
INVALID_FUNCTION_ID	7	
Busy	8	Device (or receiver) cannot answer immediately to this request for any reason i.e: already processing a request from the same or another SW pipe full ...
Unsupported	9	

0x0000 IRoot: the entry point feature

[0x0000]IRoot

featureIndex, featureType = [0]GetFeature(featureID)
ping = [1]GetFeature(pingData) /* this function is not part of the protocol itself as it is a dual HID++1.0 / HID++2.0 message **(the version of the protocol can't be inside a version of the protocol)*/**

IRoot

The entry point *feature* which is always at index 0.

GetFeature

Summary

Given a desired *feature* ID, returns its index (reference) in the feature table.

Parameters

featureID [16bits] The ID of the desired *feature.* The ID 0 is forbidden (root feature ID)

Returns

featureIndex [8bits] The one based feature index in the *feature* table, 0 if not found
featureType [2bits] This value is also provided in IFeatureSet.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
1: obsolete feature 0: active feature	1: SW hidden feature 0: SW supported feature	Reserved for Logitech internal use	n/a	n/a	n/a	n/a	n/a

Errors

No specific error

Remarks

The feature table is *one based*. Index 0 is forbidden (root index or not found value)

An **obsolete feature** is a feature that has been replaced by a newer one, but is advertized in order for older SWs to still be able to support the feature (in case the old SW does not know yet the newer one).

A **SW hidden feature** is a feature that should not be known/managed/used by standard user SW

Request

0	1	2
featureID:MSB	featureID:LSB	N/A

Response

0	1	2
featureIndex	featureType	N/A

SW Response Error Exception

none.

SW Description:

The Root feature is used by the software to determine if an unknown device is HID++ 1.0 or 2.0. The GetProtocolVersion is a feature of the RootFeature.
Once HID++ 2.0 is confirmed there are 2 main types of software implementations which will use the Root feature to interface with a device :

0x0001 IFeatureSet: Get list of device Features

[0x0001]IFeatureSet

count	=	[0]GetCount()
featureID, featureType	=	[1]GetFeatureID(featureIndex)

IFeatureSet

The feature set *feature* which enumerates features.

GetCount

Summary

Returns the number of features contained in the set, **not including the root feature.**

Parameters

None

Returns

count [8bits] The number of features in the set, not including the root feature.

Errors

No specific error

GetFeatureID

Summary

Given a feature index returns its ID

Parameters

featureIndex [8bits] The one based feature index in the feature table.

Returns

featureID [16bits] The ID of the feature

featureType [2bits] This value is also provided in IRoot.

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 7	bit 0
1: obsolete feature 0: active feature	1: SW hidden feature 0: SW supported feature	Reserved for Logitech internal use	n/a	n/a	n/a	n/a	n/a

Errors

OutOfRange [8bits] If index > Count

Remarks

An **obsolete feature** is a feature that has been replaced by a newer one, but is advertized in order for older SWs to still be able to support the feature (in case the old SW does not know yet the newer one).

A **SW hidden feature** is a feature that should not be known/managed/used by standard user

Request

0	1	2
featureIndex	N/A	

Response

0	1	2
featureID:MSB	featureID:LSB	featureType

Example

```
    IFeatureSet featureSet = Root.GetFeature(ID=1)
    for (featureIndex = 1; featureIndex <= featureSet.GetCount(); ++featureIndex)
(featureID, featureType) = featureSet.GetFeatureID(featureIndex)
if (featureType == FeatureType.Ignore) continue
yield return FeatureFactory.Create(featureID, ...)
```

SW Description:

This feature is used to enumerate the features on a device.

- Features which are flagged with the "SW hidden" features are completely ignored (skipped) unless a an override for a specific feature ID (and possible device model) becomes necessary.

Typical implementation will first query the # of features and then call GetFeatureID for each index from 1 to Count. Software will store this information in a file (cache) and will only repeat this when the cache is no longer in synch with the device (e.g. if the device firmware version has changed since the creation of the current cache).

0x0003 LONG Get FW version, build + protocol_specific_info

[0x0003]IFirmwareInfo

entityCount = [0]GetEntityCount()
entity, protocol, version, build, dynamic_FWconf, transportLayer_specific_info = [1]GetFwInfo(entity)

IFirmwareInfo

The firmware info *feature*

The firmware entities represent different codes running on the same processor (main firmware and bootloader, etc.) or on different processors.

Shall also be used by SW with FwType=0 for CacheID as information in this function represents a unique ID for a given device/build/etc...

GetEntityCount

Summary

Returns the number of firmware and hardware entities

Parameters

None

Returns

entityCount [8bits] The number of firmware entities

Errors

None

Remarks

Response

0	1	2
entityCount	N/A	

GetFwInfo

Summary

Returns the firmware version. Shall also be used by SW with FwType=0 for CacheID as information in this function represents a unique ID for a given device/build/etc...

Parameters

FwHWEntity [8bits] The firmware or hardware entity for which we want the version

Returns

FwType [4bits] The FW type parameter

enum **FirmwareType**:

Main application	= 0 (the one that talks thru HIDpp2.0)
BootLoader	= 1
Hardware	= 2 (used to know HW version)
Other	= 3..15

FwPrefix [24bits] The ASCII prefix of the firmware entity
FwVersion [16bits] The BCD version number of the firmware entity
FwBuild [16bits] The build number of the firmware entity
XX [8bits]
transportLayer_Specific_info [up to 9 bytes] Transport layer (USB, Unifying) specific info (such as UnifyingID/PID for instance)

Errors

OutOfRange [8bits] If index > entityCount

Remarks

Result in BCD (by analogy with the USB "bcdDevice" and for consistency with HID++ 1.0) [TBD]

Request

0	1	2
FwHwEntity	N/A	

Response FwType = BootLoader, Main application

0		1	2	3	4	5	6	7	8	9 ... 15
N/A	FwType	FwPrefix:char1	FwPrefix:char2	FwPrefix:char3	FwVersion:MSB	FwVersion:LSB	FwBuild:MSB	FwBuild:LSB	XX	transportLayer_Specific_info

Response FwType = Hardware

0		1	2	3	4	5	6	7	8 ... 15
N/A	2	HW Version	N/A	N/A	N/A	N/A	N/A	N/A	N/A

0x0005 Device Name and Type

[0x0005]GetDeviceNameType

nameLength = [0]GetCount()
DeviceName = [1]GetDeviceName(CharIndex)
DeviceType, deviceInterface(s) = [2]GetDeviceType()

GetDeviceNameType

get device name and type. name is also done at Unifying level.

GetCount

Summary

Returns the length of the device name that should appear in the SW. Note this does not include any termination zeros.

Parameters

None

Returns

count [8bits] The length of the device name that should appear in the SW.

Errors

No specific error

GetDeviceName

Summary

Returns a chunk of characters of the name starting from the index specified in the requested.
The size of the chunk is the entire payload of the transport packet used by the device to respond (HPPLong : 16 characters, HPPShort : 3 characters)

Parameters

CharIndex [8bits]

Returns

DeviceName Device name chunk starting at charindex.

Errors

OutOfRange (if Char Index > device name length)

Remarks

To query the device name the host first queries the length (GetCount)

SW Response Error Exception

none.

GetDeviceType

Summary

Returns device type

Parameters

None

Returns

type enum:

Keyboard = 0
RemoteControl =1
NUMPAD = 2
Mouse = 3
Touchpad = 4
Trackball = 5
Presenter = 6
Receiver = 7

Errors

None

0x1000 Battery Unified Level Status

[0x1000]BatteryLevelStatus

BatteryLevel	= [0]GetBatteryLevelStatus()
LevelList[]	= [1]GetBatteryCapability()

Event

BatteryLevel	= [0]BatteryLevelStatusBroadcastEvent()

GetBatteryLevelStatus

Returns battery status to SW

BatteryLevelStatusBroadcastEvent

Reports battery status to SW spontaneously

Report

BatteryDischargeLevel [8bits] The current level expressed in %. 0 means unknown.

BatteryDischargeNextLevel [8bits] The next level to be reported in % as the device battery discharges. If the current level is the lowest level or if the device is charging this value is set to 0.

BatteryStatus [8bits] Enum :

0 = discharging (in use)
1 = recharging
2 = charge in final stage
3 = charge complete
4 = recharging below optimal speed
5 = invalid battery type
6 = thermal error
7 = other charging error
8..255 : invalid

Remarks

Report

byte 0	byte 1	byte 2	byte 3..15
BatteryDischargeLevel	BatteryDischargeNextLevel	BatteryStatus	n/a

SW Response Error Exception

next level > current level

Current level in [0..100] range

BatteryStatus in [0..6] range

GetBatteryCapability

Summary

Returns the static capability information about the device.

Parameters

None

Returns

NumberOfLevels [8bits] The number of levels the device is capable of reporting to the SW. Min : 2 - Max : 100.

If number of levels < 10 or if mileage is disabled then report are mapped to 4 levels this way.

0%->10%	critical
11%->30%	low

31%->80% good
81%->100% full
i.e. to report battery low, FW send 25%, to report battery good, FW send 50%.

Flags [8bits]
bit 0 : Disable battery OSD
bit 1 : Enable mileage calculation. This flag is ignored by the SW if NumberOfLevels is less than 10.
bit 2 : Rechargeable
bit 3 - 7 : must be set to zero.

NominalBatteryLife [16bits] Defined as Duration Type . As of today, this is only used when battery mileage is enabled.
This could be the advertized battery life.

BatteryCriticalLevel [8bits] in % only used when battery mileage is enabled.

Battery Levels	Value
InvalidBatteryLevel not-Rechargeable	0x00
BatteryLow	0x18
BatteryGood	0x38
BatteryFull	0x58
BatteryError	0x68
BatteryLevelUnknown	0x69
WrongBatteryType	0x6B
BatteryNotCharging	0x6C
BatteryChargingUnknown	0x6D
BatteryCharging	0x6E
BatteryChargingSlow	0x6F
BatteryChargingFast	0x70
BatteryChargingComplete	0x71
BatteryChargingError	0x72

0x1B00 KBD reprogrammable Keys and MSE buttons (Rev1)

[0x1B00]SpecialKeysMSEButtons

ctrlIDCount = [0]GetCount()
ctrlIDIndex+flags list = [1]GetCtrlIDInfo(ctrlIDIndex)

Event

ctrlIDIndexPressedList = [0]ControlIDBroadcastEvent()

SpecialKeysMSEButtons

manages all SpecialKeys listed here:

Control ID table

Control Task Assignments

This feature is the engine enabling to manage of all FnKeys, HotKeys & MSE buttons.

GetCount

Summary

Returns the number of Keys and/or MSE Buttons defined.

Parameters

None

Returns

count [8bits] The number of Control IDs in the list.

Errors

No specific error

GetCtrlIDInfo

Summary

Returns a ctrl_ID and ctrl_Task indexes pointing towards tables defined below.
Concretely, let's assume a device with 7 controls which I will call A,B,C,D,E,F,G :
- GetCtrlIDInfo(0) returns A
- GetCtrlIDInfo(3) returns D
- GetCtrlIDInfo(6) returns G

Parameters

ctrlIDIndex[8bits]

Returns

ctrl_ID_Index [16bits] ctrl_ID_Indexes

ctrl_Task_Index [16bits] ctrl_Task_Indexes

flags [8bits] field

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
			Reprogrammable: 0: NO 1: YES	Affected by FnToggle: 0: NO 1: YES	Hotkey not a Standard key on KBDs. Standard keys have a "code" from 1 to 126.	FnKey (F1 to F12 or F16)	MSE Button*(1)

*(1): Use this bit for any function trigged by a **mouse** button, so it will appear in the mouse TAB of SetPoint. This includes volume +/-, Search, etc...

For bits 3 to 0 only the following values are valid : **0001** (Mouse button), **0010** (Fn Key), **0100** (Hot key), **1010** (Fn Key invertible).
Note : A control ID wich would have the re-programmable bit to NO and the task ID "generic control" 72 () would be equivalent to not report this control ID in the list.

Errors

OutOfRange (if CtrlID Index > keymsebuttoncount)

Request

byte 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	byte 15
Control ID index	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Response

byte 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	byte 15
Sent/FW Control_ID MSB	Sent/FW Control_ID LSB	Desired/ SW Control_Task MSB	Desired/ SW Control_Task LSB	Flags	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

SW Response Error Exception

none

ControlIDBroadcastEvent

Summary

This event is used to only report the following specific control IDs for which the native report is not sufficient or does not exist :

Control ID's which do not have a make/break type of HID report (LC type. See :). Note that in this case the device must report **both** the linear control HID usage and this notification.

Control ID's mapped to generic HID buttons (Usage Page 0x09) **except if they are generic mouse buttons** or if the device does not have any HID++ interface.

e.g. :
If Gadgets (d33) is used on a Unifying device or a corded device which has HID++ collections it should report 0x0021 (d33) in the ControlIDBroadcastEvent notification.
If Gadgets (d33) is used on a USB corded keyboard which does **NOT** have any HID++ collection (standard PID) then it should report this key as generic button **0x019F**

All other control IDs should **NOT** use this notification and always be reported in their respective HID interface as defined in the control ID specification table.

Event data

The report contains an array of the applicable (see above) **control IDs** of the currently pressed controls (keys/buttons). Each control ID is encoded on 2 bytes in big endian (MSB first).
When a control is released a report where this control ID is no longer present. Inactive value is 0x0000.
A maximum of 4 special control IDs can be pressed simultaneously.
All bytes outside of the array should be ignored.

The example below shows a sequence of reports with 2 control IDs pressed and released :

Control 1 ID pressed (**make**)

byte 0	1	2	3	4	5	6	7	bytes 8..15
Control 1 ID MSB	Control 1 ID LSB	0	0	0	0	0	0	n/a

Control 2 ID pressed (**make**) and Control 1 ID still pressed.

byte 0	1	2	3	4	5	6	7	bytes 8..15
Control 1 ID MSB	Control 1 ID LSB	Control 2 ID MSB	Control 2 ID LSB	0	0	0	0	n/a

Control 1 ID released (**break**) and Control 2 ID still pressed.

byte 0	1	2	3	4	5	6	7	bytes 8..15
Control 2 ID MSB	Control 2 ID LSB	0	0	0	0	0	0	n/a

Control ID 2 released (**break**)

byte 0	1	2	3	4	5	6	7	bytes 8..15
0	0	0	0	0	0	0	0	n/a

Control ID	**Description**	**HID**	**Suggested Task ID**
1 / 0x1	Volume Up	(Consumer Control) Usage : 0x0C / Usage : **0x00E9** (Volume Increment)	1 Volume Up
2 / 0x2	Volume Down	(Consumer Control) Usage : 0x0C / Usage : **0x00EA** (Volume Decrement)	2 Volume Down
3 / 0x3	Mute	(Consumer Control) Usage : 0x0C / Usage : **0x00E2** (Mute)	3 Mute
4 / 0x4	Play/Pause	(Consumer Control) Usage : 0x0C / Usage : **0x00CD** (Play/Pause)	4 Play/Pause
5 / 0x5	Next	(Consumer Control) Usage : 0x0C / Usage : **0x00B5** (Scan Next Track)	5, Next with fast forward on long press /96 Next
6 / 0x6	Previous	(Consumer Control) Usage : 0x0C / Usage : **0x00B6** (Scan Previous Track)	6 Previous with rewind on long press /97 Previous
7 / 0x7	Stop	(Consumer Control) Usage : 0x0C / Usage : **0x00B7** (Stop)	7 Stop

Task ID (dec)	Task Name	Description
1	Volume Up	Volume + (Increases volume on all connected playback devices)
2	Volume Down	Volume - (Decreases volume on all connected playback devices)
3	Mute	Toggle b/w Volume Mute and UnMute state (Displays OSD)
4	Play/Pause	Toggles between play and pause state in SetPoint supported players. ("Reads Players.ini which contains the commands to be sent to each player (Standard APP Command , Keyboard shortcut, or Button events.")
5	Next /Fast forward	Next track and fast forward on long press (see task 96)
6	Previous /rewind	PreviousTrack and rewind on long press (see task 97)
7	Stop	Puts the player in STOP state. (Reads Players.ini which contains the commands to be sent to each player (Standard APP Command , Keyboard shortcut, or Button events.)
8	Application Switcher	Activates the application switcher.

0x1D4B Wireless Device Status

[0x1D4B]WirelessDeviceStatus

Event

Status, Request, Reason = [0]WirelessDeviceStatusBroadcastEvent()

WirelessDeviceStatusBroadcastEvent

Sent by the device after a power-on reset

Report

Status	[8bits]	0x00	no status
		0x01	reconnection
		0x02..0xFF	reserved
Request	[8bits]	0x00	no request
		0x01	software reconfiguration needed
		0x02..0xFF	reserved
Reason	[8bits]	0x00	unknown
		0x01	power-switch activated
		0x02..0xFF	reserved

Remarks

This notification is always enabled since it must be sent by a device after a power-on reset.

Report

0	1	2
Status	Request	Reason

SW Response Error Exception

none.

HID++2.0 Device transaction example

```
                        31
utils             95    16:27:59  496 T ****START OF SCRIPT RUN MODE****
                  157             529 T ****UNITTEST MODE: False****
SamarkandDevice   92              544 1 test: SamarkandDevice
                                  545 1 start with USB VID: 0x046D, PID: 0xC52B
                        31
                        16:28:00  824 1 end!
ListFWs           28              874 4 Choosing a HID++ device
                  33              896 X 10 00 00 10 00 00 AA ping!
                                  900 X 10 01 00 10 00 00 AA ping!
                                  904 X 10 02 00 10 00 00 AA ping!
                                  908 X 10 03 00 10 00 00 AA ping!
                                  912 X 10 04 00 10 00 00 AA ping!
                                  918 X 10 05 00 10 00 00 AA ping!
                                  922 X 10 06 00 10 00 00 AA ping!
                                  926 X 10 07 00 10 00 00 AA ping!
                                  945 X 10 08 00 10 00 00 AA ping!
                  36              950 R (150)10 00 8F 00 10 08 00
                                  973 R (1000)10 01 8F 00 10 09 00
                                  974 R (1000)10 03 8F 00 10 09 00
                                  977 R (1000)10 04 8F 00 10 09 00
                                  977 R (1000)10 05 8F 00 10 09 00
                                  979 R (1000)10 06 8F 00 10 09 00
                                  979 R (1000)10 07 8F 00 10 08 00
                                      R (1000)11 02 00 10 02 00 AA 00 00 00 00 00 00
                                  979 00 00 00 00 00 00 00
                  28              982 1 Found a HID++2.0 device
                                      X 10 02 00 00 00 03 00 HID++2.0 get address of
                  27              987 0x0003 FWinfo feature
                  36              987 R (1000)10 08 8F 00 10 08 00
                        31            R (1000)11 02 00 00 03 00 00 00 00 00 00 00 00
                        16:28:01  028 00 00 00 00 00 00 00
                  28              029 1 HID++2.0 getFeatureInfo[2] has index=3
                                      X 10 02 03 10 00 00 00 HID++2.0 get Main FW
                  27              033 name and build
                                      R (1000)11 02 03 10 00 52 51 4B 40 00 00 08 00
                  36              068 40 0D 00 00 00 00 00
                  28              078 1 HID++2.0 has fw=RQK40.00
                        31
__main__                16:28:02  099 4 RQK40.00 Build: 2 Has been selected.
                                      X 10 02 00 1E 00 00 AA GetProtocolVersion of
SamarkandDevice   85              105 device[2]
                                      R 11 02 00 1E 02 00 AA 00 00 00 00 00 00 00 00
                                  137 00 00 00 00 00
                  44              162 2 ****device[2] is HID++2.0
                  51              169 X 10 02 00 0E 00 03 00 GetFeature(0x0003)
                                  196 R 11 02 00 0E 03 00 00 00 00 00 00 00 00 00
```

		00 00 00 00 00
117	197	1 GetFeature(0x0003) @ index: 3
124	203	3 [00] is in [00 40 80 C0] -> True --- Has a Valid feature Type!
37	209	X 10 02 03 1E 00 00 00 GetFwInfo
	236	R 11 02 03 1E 00 52 51 4B 40 00 00 08 00 40 0D 00 00 00 00 00
41	236	1 Device[2] is HID++2.0, build number : B0008