Copyright 2014, Nordic Semiconductor ASA

Nordic Semiconductor Sniffer API Guide

Version 0.2

The Sniffer API guide provides the documentation of the Python API used to interface with the nRF Sniffer for Bluetooth low energy. The nRF Sniffer is available for download from mypage at nordicsemi.com on purchase of the nRF51822, nRF51422 and nRF8001 development kits. The Python API documented is currently available only for Windows. The intent of this document is to support the porting of the Sniffer API to non-windows platforms like OS X and Linux.

Revision History

Revision	**Changes**
0.1	Initial version
0.2	Added description of LED and GPIO.
0.3	Updated documentation to reflect API changes after 0.9.7

Copyright 2014, Nordic Semiconductor ASA

Introduction

The Sniffer API is a Python API that allows scripted use of the Nordic Semiconductor BLE Sniffer. It allows discovery of devices and sniffing of a single device. It provides access to all the BLE packets received by the sniffer and the devices discovered.

The sniffer consists of three parts as seen in Figure 1, where the API replaces the console app as the controller and hub of communication.

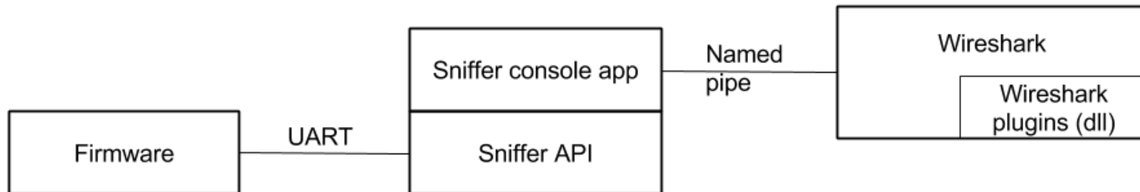


Figure 1 - The parts of the sniffer.

The Wireshark plugin code is included in the API.

Dependencies

The API has been developed using Python 2.7.6 32 bit. 64 bit is untested but might work. The API also requires one third party Python library:

1. Pyserial (cross platform) version 2.7. Get the installer if you are on Windows.
   http://pyserial.sourceforge.net

In addition, you must get nRF Sniffer version 0.9.7, and make sure it connects to the firmware.

See the Sniffer User Guide included with the nRF Sniffer for more information.

Using the Sniffer API

Getting Started

1. Install dependencies.
2. Include the SnifferAPI folder in your Python project.
3. Import the API with
   ```
   from SnifferAPI import Sniffer
   ```
4. Instantiate the Sniffer class with e.g.
   ```
   mySniffer = Sniffer()
   ```
5. Start the Sniffer with
   ```
   mySniffer.start()
   ```

`example.py` is an example program with explanations in the comments.

Overview

The API consists of 5 classes in 3 files: The Sniffer class in Sniffer.py, the DeviceList and Device classes in Devices.py, and the Packet and BlePacket classes in Packet.py. The

Copyright 2014, Nordic Semiconductor ASA

exceptions in Exceptions.py are also part of the API. The entry point for the API is the Sniffer class (retrieve packets and devices through the methods in Sniffer). The last pages of this document (and also the documentation.html file) contain a complete documentation of the API.

An overview of the levels below the Sniffer module

Object/Module hierarchy

During normal operation, the Sniffer object interfaces only to the SnifferCollector object which acts as a hub for the flow of packets. The SnifferCollector object reads packets from UART through its PacketReader object, and sends packets over named pipe to Wireshark. It also stores all packets in a capture (.pcap) file through its CaptureFileHandler object, and keeps an internal buffer of packets. In addition, the SnifferCollector object keeps a list of devices which are advertising in the vicinity.

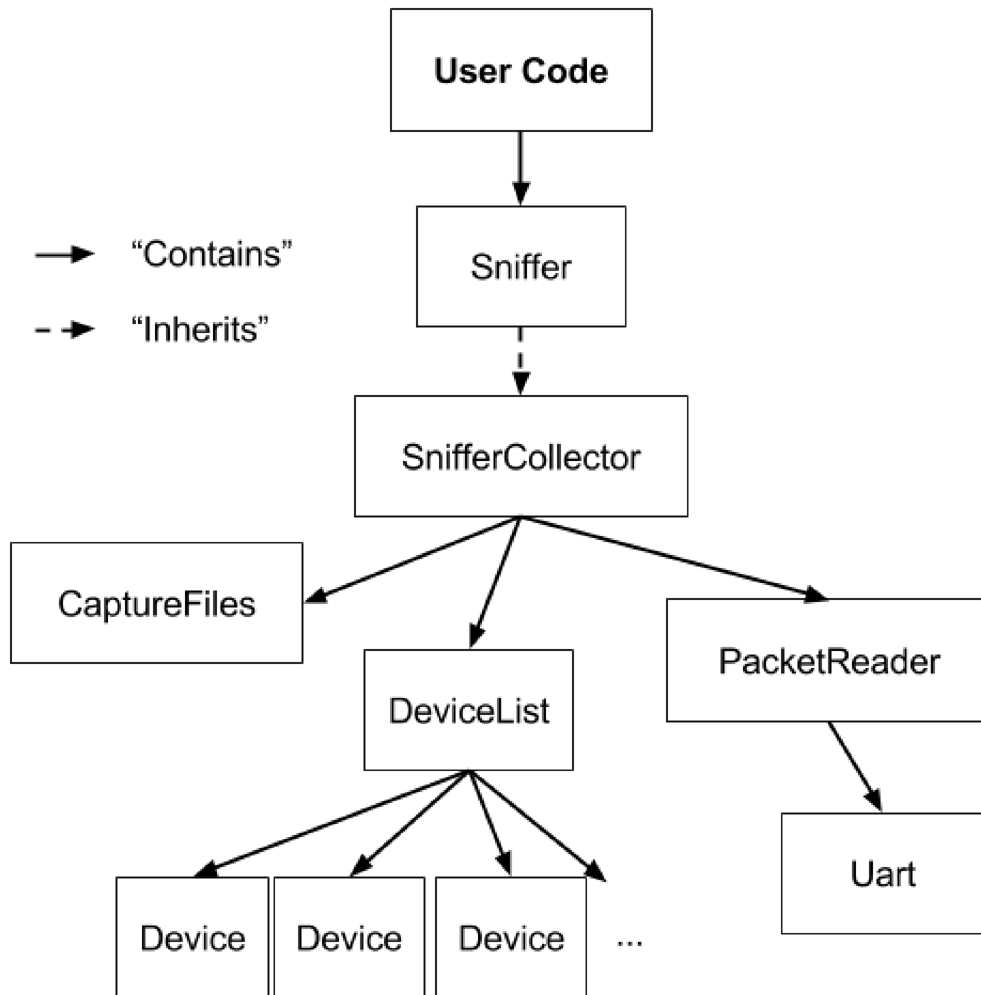


Figure 2- Object hierarchy behind the Sniffer API

Copyright 2014, Nordic Semiconductor ASA

Figure 3- The flow of packets through the API.

Note: Command packet flow from the SnifferCollector to the UART is not represented in the above diagram.

Threads of operation

The Sniffer system contains 3 separate threads which are running in addition to the main context (user thread). They are:

1. The Pipe thread which is used to connect the named pipe dynamically.
2. The LogFlusher thread which regularly flushes the log to file.
3. The Sniffer thread. This is the main thread which handles everything else, including the flow of packets described above.

OS specific code

The API should not contain any OS specific code. The modules that previously had OS specific code have been removed in this version of the API.

Establishing a connection between the API and the firmware

As explained below, the firmware sends PING_RSP packets in the SCANNING state. The Sniffer constructor can take the port number of the firmware as an argument. In this case, the API connects blindly to it. If no port is provided, the API opens all COM ports on the computer in succession and listens for PING_RSP packets to locate the correct port. When the PING_RSP packet is not received on a COM port, it closes the COM port.

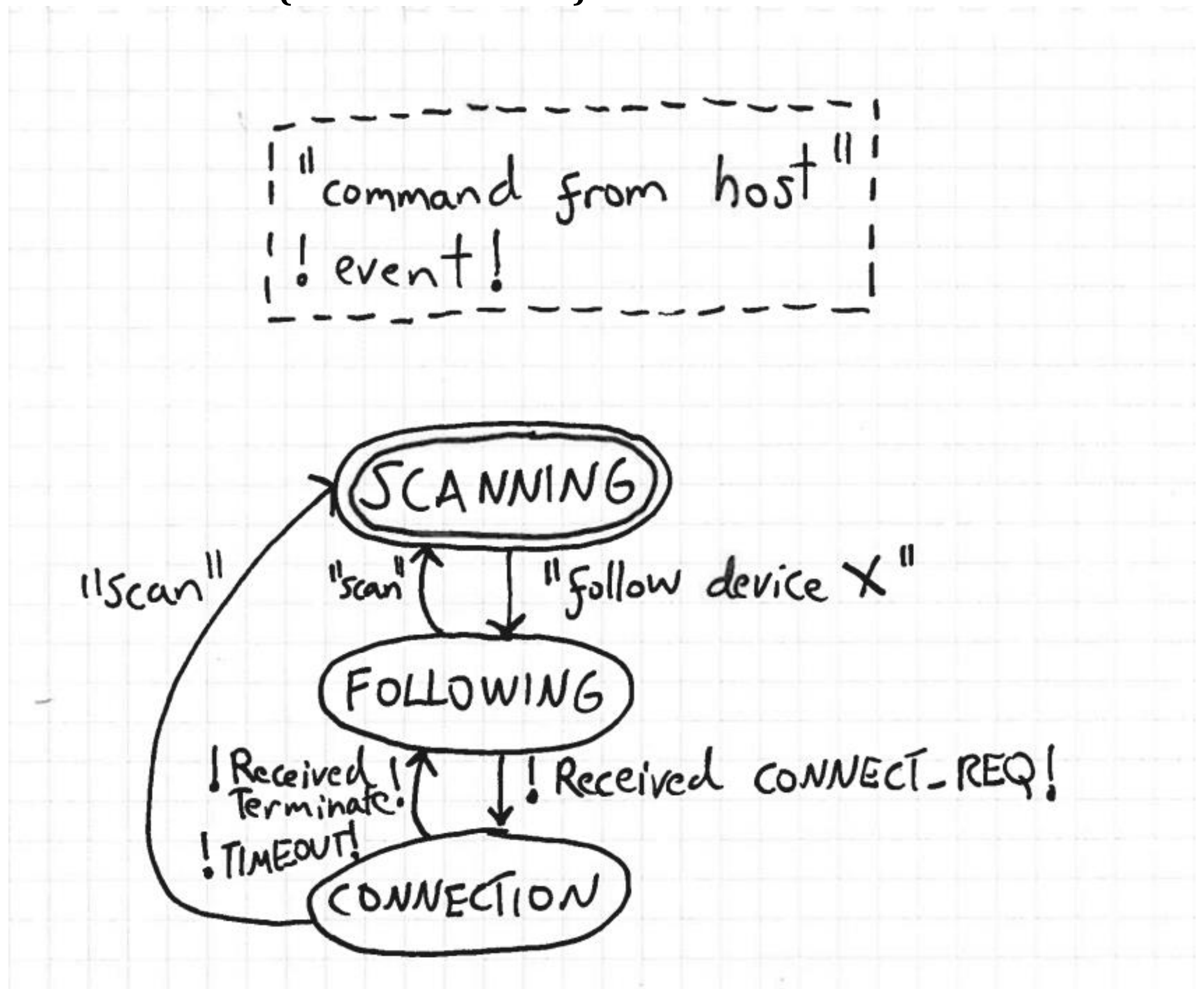
Copyright 2014, Nordic Semiconductor ASA

Appendices

1. State change description
2. API documentation (also in documentation.html)
3. Description of UART protocol (also in sniffer_uart_protocol.xlsx)

Copyright 2014, Nordic Semiconductor ASA

Firmware States (nRF Sniffer v0.9.6)



SCANNING (Initial state):

- Scans advertiser packets.
- The sniffer will send a PING_RSP each 75ms to the host.

State change: If the sniffer received a "follow device X" command, it will go to the FOLLOWING state.

FOLLOWING:

- Only packets from device X will be received.
- All packets sent by device X will be received.
- All SCAN_REQ packets directed to device X and corresponding SCAN_RSP packets will be picked up.

Copyright 2014, Nordic Semiconductor ASA

- All CONNECT_REQ packets directed to device X will also be picked up.

State change:

- If the sniffer receives a CONNECT_REQ packet, it will go to the CONNECTION state.
- If the sniffer received a "scan" command, it will go to the SCANNING state.

CONNECTION:

- The sniffer will follow the connection.
- All packets in the connection will be received.

State change:

- If a timeout occurs (no packets received for about 30 seconds) the sniffer will go to the FOLLOWING state.
- If one of the devices in the connection terminating the connection the sniffer will go to the FOLLOWING state.
- If the sniffer received a "scan" command, it will go to the SCANNING state.

LED Configuration (only valid for PCA10001)

State	LED0	LED1
SCANNING	OFF	Toggle when packet received
FOLLOWING	Toggle when packet received	OFF
CONNECTION	ON	Toggle when packet received

GPIO Behavior (only valid for PCA10001)

PIN	LOW – HIGH	HIGH - LOW
4	Finished receiving advertise packet from device being followed.	Enable RX for receiving CONNECT_REQ to followed device.
5	Start radio for receiving anchor point of connection event, ramp up required	Finished receiving ADDRESS bytes of anchor point in connection event.

Sniffer

The entry point for the API.

Field	Type	Description
missedPackets	int	The number of missed packets over the UART, as determined by the packet counter in the header. Derived using the packetCounter field.
packetsInLastConnection	int	The number of packets which were sniffed in the last BLE connection. From CONNECT_REQ until link loss/termination.
connectEventPacketCounterValue	int	The packet counter value of the last received connect request.
inConnection	bool	A boolean indicating whether the sniffed device is in a connection.
currentConnectRequest	Packet	A Packet object containing the last received connect request.
state	int	The internal state of the sniffer. States are defined in SnifferCollector module. Valid values are 0-2.
portnum	int or string	The COM port of the sniffer hardware. During initialization, this value is a preset.
swversion	int	The version number of the API software.
fwversion	int	The version number of the sniffer firmware.
Function	**Type**	**Description**
__init__(portnum)	Sniffer	Constructor for the Sniffer class. The optional argument "portnum" is a string with the name of the port the sniffer board is at, e.g. "COM17". If not provided, the API will locate it automatically, but this takes more time.
start()	void	Starts the Sniffer thread. This call must be made (once and only once) before using the sniffer object.
getPackets(number)	List<Packet>	Get [number] number of packets since last fetch (-1 means all). Note that the packet buffer is limited to about 80000 packets.
getDevices()	DeviceList	Get a list of devices which are advertising in range of the Sniffer.
follow(device, followOnlyAdvertisements)	void	Signal the Sniffer firmware to sniff a specific device. If followOnlyAdvertisements is True, the sniffer will not sniff a connection, only advertisements from the followed device.
scan()	void	Signal the Sniffer to scan for advertising devices by sending the REQ_SCAN_CONT UART packet. This will cause it to stop sniffing any device it is sniffing at the moment.
sendTK(TK)	void	Send a temporary key to the sniffer for use when decrypting encrypted connections. TK is a list of 16 ints, each representing a byte in the temporary key. TK is on big-endian form.
setPortnum(portnum)	void	Set the preset COM port number. Only use this during startup. Set to None to search all ports.
doExit()	void	Gracefully shut down the sniffer threads and connections.

Device

Class representing a BLE device from which the sniffer has picked up data.

Field	Type	Description
address	List< int >	A list representing the device address of this device: [int, int, int, int, int, int]
txAdd	bool	A boolean representing whether the device address is public (False) or random (True).
name	string	A string containing the name (short or complete) of the device.
RSSI	int	An int representing the approximate RSSI value of packets received from this device.

DeviceList

A class representing a list of devices. Used to simplify extraction of devices using BLE metadata.

Function	Type	Description
find(id)	Device	Find a device in this DeviceList using either name or address. Returns None if no device is found.
remove(id)	Device	Remove a device from this DeviceList. Argument "id" has same format as in find.
append(Device)	void	Append a Device to the device list.
index(Device)	int	Returns the index of the provided Device.
getList()	List<Device>	Returns a list of the Devices in this DeviceList.

Packet

Represents the UART packet sent by the sniffer to the host.

Field	Type	Description	
headerLength	int	The length of the UART header.	UART header
payloadLength	int	The length of the UART payload.	
protover	int	The UART protocol version used.	
packetCounter	int	Unique (16 bit) packet identifier which increments for each packet sent by the sniffer.	
id	int	Identifier telling what type of packet this is. See UART protocol document.	
bleHeaderLength	int	Length of the NRF_BLE_PACKET header.	NRF_BLE_PACKET header
crcOK	bool	Was the CRC received by the sniffer OK.	
micOK	bool	Is the message integriy check OK. Only relevant in encrypted state.	
direction	bool	Only relevant during connection. True -> Master to Slave, False -> Slave to Master	
encrypted	bool	has the packet been encrypted.	
channel	int	Which channel was the packet picked up from [0 - 39]	
RSSI	int	The RSSI value reported by the sniffer. NOT PRECISE. Real value is the negative of this value.	
eventCounter	int	The eventcounter of the packet in the connection. Only relevant for packets in a connection.	
timestamp	int	Microseconds from the end of the last packet to the start of this one.	
blePacket	BlePacket	The blePacket contained within this packet.	Other
packetList	List< int >	The entire UART packet as sent by the sniffer (with the exception of a padding byte which is removed).	
OK	bool	Is the error detection of the attached BLE packet OK?	
payload	List< int >	List containing the UART payload as bytes.	
txADD	bool	Is the address public or random? True -> Random, False -> Public. Only relevant for advertisement packets.	
version	int	The firmware version of the sniffer. Only sent in PING_RESP packets.	

BlePacket

Represents the BLE packet received over the air by the sniffer.

Field	Type	Description
accessAddress	List< int >	A list of bytes representing the access for this packet.
advType	int	The advertisement type field.
advAddress	List< int >	The advertising address.
name	string	The value of the localname property of the ble packet.
payload	List< int >	The entire BLE payload (not including access address and header fields).
length	int	The value of the length field of the BLE PDU

Exceptions

The exceptions raised by the API.

Exception	Description
SnifferTimeout	UART read time out.
UARTPacketError	UART SLIP parsing error.
InvalidPacketException	Other UART parsing error.

Packet format: {HEADER} {PAYLOAD}

Packet header: **[HLEN] [PLEN] [PROTOVER] [PC0][PC1] [ID]**
HLEN: Header length
PLEN: Payload length
PROTOVER: UART protocol version used
PC: Packet Counter (LSB)
ID: Packet type, see below

SLIP encoding:

Characters:	Characters when escaped:
SLIP_START: 0xAB	SLIP_ESC_START: 0xAC
SLIP_END: 0xBC	SLIP_ESC_END: 0xBD
SLIP_ESC: 0xCD	SLIP_ESC_ESC: 0xCE

UART Packet IDs (grey fields: currently not in use; beige: not yet implemented)

Byte value [ID]	Name	Direction	Payload formats:	Function
0x00	REQ_FOLLOW	Host->Sniffer	{ADDRESS} [ADDR_TYPE] [FOLLOW_ONLY_ADVERTISEMENTS]	Tell the Client to only send packets recieved from a specific address.
0x01	EVENT_FOLLOW	Sniffer->Host	[]	Client tells the host that it has entered the FOLLOW state.
0x02				
0x03				
0x04				
0x05	EVENT_CONNECT	Sniffer->Host	[]	Client tells the host that someone has connected to the unit we are following
0x06	EVENT_PACKET	Sniffer->Host	{NRF_BLE_PACKET}	Client tells the host that it has recieved a packet
0x07	REQ_SCAN_CONT	Host->Sniffer	[]	Host tells the client to scan continously and hand over the packets asap.
0x08				
0x09	EVENT DISCONNECT	Sniffer->Host	[]	Client tells the host that the connected address we were following has recieved a disconnect
0x0A				
0x0B				
0x0C	SET_TEMPORARY_KEY	Host->Sniffer	{TEMPORARY_KEY}	Specify the temporary key to use on encryption (for OOB and passkey)
0x0D	PING_REQ	Host->Sniffer	[]	
0x0E	PING_RESP	Sniffer->Host	{FW_VERSION}	
0x13	SWITCH_BAUD_RATE_REQ	Host->Sniffer	[BAUD0] [BAUD1] [BAUD2] [BAUD3]	
0x14	SWITCH_BAUD_RATE_RESP	Sniffer->Host	[BAUD0] [BAUD1] [BAUD2] [BAUD3]	
0x17	SET_ADV_CHANNEL_HOP_SEQ	Host->Sniffer	[N_CHANS] [CHAN0] [CHAN1] [CHAN2]	Tell the sniffer which order to cycle through the channels when following an advertiser.
0xFE	GO_IDLE	Host->Sniffer	[]	When receiving this, the sniffer should stop sending UART traffic, and listen for new

NRF_BLE_PACKET

Packet format: [HEADER] [PAYLOAD]

Header [HLEN] [FLAGS] [CHANNEL] [RSSI] [EC0] [EC1] [TD0] [TD1] [TD2] [TD3]

FLAGS (1 byte in total) [][][][][MICOK][ENCRYPTED][DIR][CRCOK]

CRCOK -> Was CRC Ok during transmission?
DIR -> Direction of the packet (0: Slave -> Master. 1: Master -> Slave)
Channel -> The channel index being used
EC -> Event Counter
TD -> TimeDiff: Delta from previously recieved packet

LEGEND:
[Square brackets denote single bytes]
{Curly brackets denote multiple bytes}

NRF_BLE_PACKET Payload (ordinary BLE Packet)

{AA x 4} [HEADER] [LEN] [PADDING] {PAYLOAD x LEN} {CRC x 3}

Note: Padding byte is added by radio and is not received on air. It should be removed after reception on UART.

BAUD RATE SWITCHING PROCEDURE (NOT YET IMPLEMENTED)
Host sends SWITCH_BAUD_RATE_REQ with proposed baud rate
Sniffer responds (SWITCH_BAUD_RATE_RESP) with proposed baud rate (must be host's proposal if this is possible)
If host and sniffer propose same baud rate, baud rate is considered changed, and both parties will configure hardware.

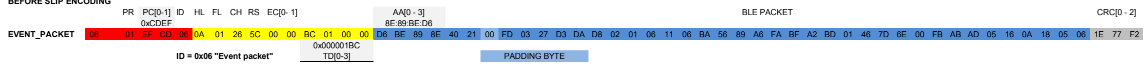
If sniffer proposes different baud rate, host may retry with another baud rate (must be sniffer's proposal if this is possible)
Neither party can propose the same baud rate twice.

SLIP ENCODING PROCEDURE
Add a SLIP_START to encoded packet

For each byte in unencoded packet, do the following:
- If the byte is not equal to SLIP_START, SLIP_END, or SLIP_ESC, add it to encoded packet.
- Otherwise, replace it with a SLIP_ESC followed by the corresponding escaped character (SLIP_ESC_START, SLIP_ESC_END, or SLIP_ESC_ESC.)

Finally: add a SLIP_END to encoded packet

BEFORE SLIP ENCODING



AFTER SLIP ENCODING