Brief

Name: nanoLambda NSP32 Python API for RPi

Type: API

Version: 1.0.0

Language: Python

Platform: Raspberry Pi (RPi)

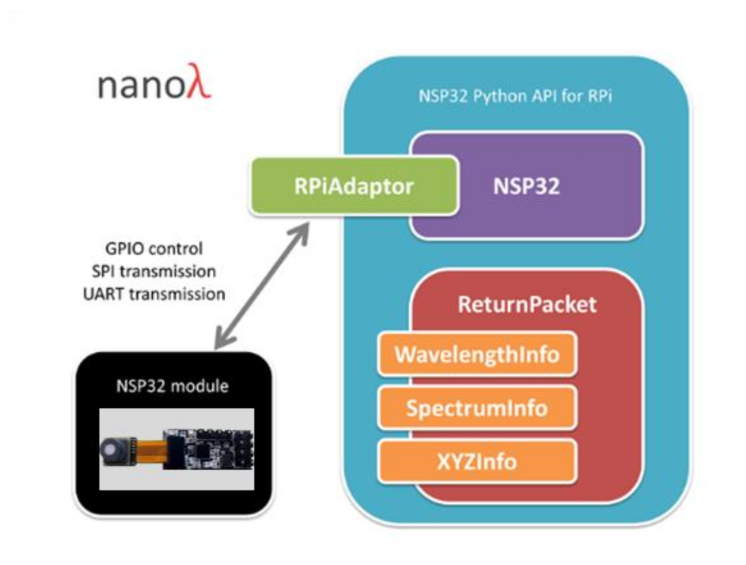
Introduction

The API is designed for runing on Raspberry Pi applications coded in Python. By using this API, users can easily control NSP32 module by high level function calls, without dealing with the raw packet bytes and timing sequences.

Note: General concepts are illustrated in NSP32 datasheet. Please see datasheet in advance.

Architecture & Concepts

1) API architecture



Note: different classes/structs and their corresponding source files are listed in different colored blocks.

The architecture contains two major parts.

i) Controller and adaptor

- NSP32 class is the main controller that deals with commands, packets, timing sequences, error detections, and other flow logic.
- RPiAdaptor class deals with RPi specific behaviors, such as GPIO pin control, SPI transmission, UART transmission.
- NSP32 class interacts with NSP32 module through RPiAdaptor. NSP32 creates and uses RPiAdaptor object internally. Users don't create any RPiAdaptor object by their own.

ii) Data

ReturnPacket class encapsulates the return packet received from NSP32 module. We also define three classes (which are placed in orange blocks in the picture) to encapsulate wavelength, spectrum, and XYZ data. Users can extract these class objects from ReturnPacket object, and retrieve their

interested information.

Example:

```
pkt = nsp32.GetReturnPacket()       # where "nsp32" is a NSP32 object, and pkt is a ReturnPacket object
info = pkt.ExtractSpectrumInfo()    # info is a SpectrumInfo object
data = info.Spectrum                # data is a float tuple
```

Please refer [/examples/] examples for complete demonstration.

2) Synchronous (Sync.) and asynchronous (async.) commands

For NSP32, some commands can be done immediately, and we can fetch the results right away (e.g. CMD_GET_SENSOR_ID). We call these synchronous (sync.) commands. However, other commands are time consuming, and we have to come back later to fetch the results (e.g. CMD_ACQ_SPECTRUM). We call these asynchronous (async.) commands.

Sync. commands are:

i) CMD_HELLO
ii) CMD_STANDBY
iii) CMD_GET_SENSOR_ID
iv) CMD_GET_WAVELENGTH

Async. commands are:

i) CMD_ACQ_SPECTRUM (and fetch the result by CMD_GET_SPECTRUM)
ii) CMD_ACQ_XYZ (and fetch the result by CMD_GET_XYZ)

Note:

For async. commands like CMD_ACQ_SPECTRUM, API handles the complete cycle (from starting acquisition to fetch results) inside. So you will see only NSP32.AcqSpectrum() function, but no NSP32.GetSpectrum() function. Due to this reason, if you investigate the FUNCTION_CODE byte in the return packet of NSP32.AcqSpectrum(), you'll find the byte is CMD_GET_SPECTRUM instead of CMD_ACQ_SPECTRUM.

Please refer [/examples/] examples to see how to use sync. and async. commands respectively with the API.

Development Tool Recommendation

Python 3.5 or above (Python 2 doesn't work)

The API utilizes the following modules, please make sure they are installed under your environment.

1) RPi.GPIO [https://pypi.org/project/RPi.GPIO/]
2) spidev [https://pypi.org/project/spidev/]
3) pySerial [https://pypi.org/project/pyserial/]

How to Use

1) Copy [/src/NanoLambdaNSP32.py] to your project folder.
2) Import "NanoLambdaNSP32" module and write codes (please refer our examples).

API Features

1) Auto checking: Upon NSP32 module wakeup / reset, a series of internal checking procedure is performed. If any malfunction is detected, NSP32 module will not generate the "ready trigger". If you are using our API for RPi, upon calling NSP32.Init() or NSP32.Wakeup(), the API also performs a SPI / UART check (by sending CMD_HELLO). If the check fails, API will reset NSP32 module and try again until successful.
2) Auto wakeup: NSP32 module has two power modes: "active mode" and "standby mode". In standby mode, no commands would be accepted. If you are using our API for RPi, it auto checks NSP32 module's current mode before sending commands, and auto wakes it up if necessary.

   Note: When power consumption is in concern, the best practice is to standby NSP32 module if you don't need it for a while.
3) Packet error detection and auto retry: The API monitors and validates each return packet. In case a packet error is detected (due to transmission error), API will automatically resend the command until successful.
4) Both SPI and UART are supported: By simply specifying the desired data channel (SPI or UART) when creating NSP32 object, the API can handle the transmission details of both channel types.

   Example:
   ```
   nsp32 = NSP32(PinRst, PinReady, DataChannelEnum.Spi)    # use SPI
   ```
   or
   ```
   nsp32 = NSP32(PinRst, PinReady, DataChannelEnum.Uart)   # use UART
   ```

5) Both return packet raw bytes and easy-to-use data objects are available: In most

cases, users extract their desired information (data object) from the return packet by calling ReturnPacket.Extract...() functions. However users can also use ReturnPacket.PacketBytes property to access the return packet raw bytes if needed.

API Reference

1. html version: [/doc/reference_html/index.html]
2. pdf version: [/doc/reference.pdf]