Brief

Name: nanoLambda NSP32 C/C++ API for MCU

Type: API

Version: 1.0.0

Language: C/C++

Platform: MCU

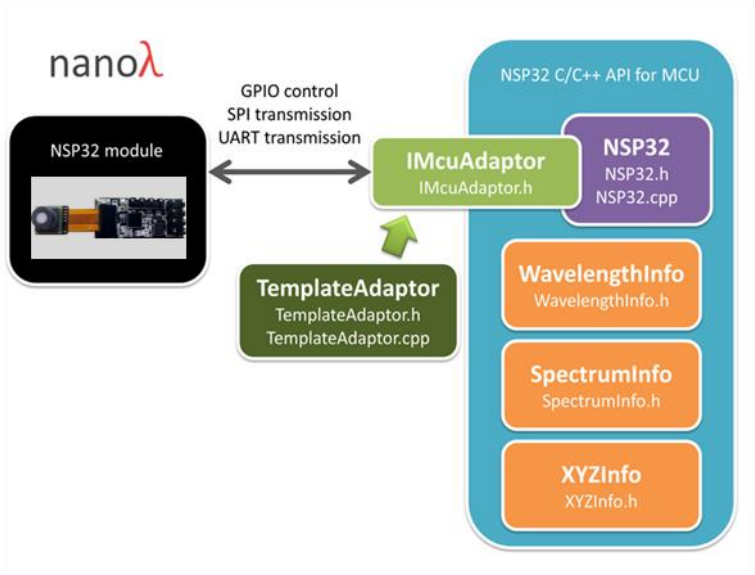
Introduction

The API is designed for use on general MCU applications coded in C/C++. 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.
- IMcuAdaptor class (abstract class) acts as an interface that defines MCU dependent behaviors, such as GPIO pin control, SPI transmission, UART transmission.
- NSP32 class interacts with NSP32 module through IMcuAdaptor.
  Once we implement an adaptor for a certain MCU (e.g. ArduinoAdaptor), NSP32 class can thus work on that MCU through the specific adaptor.

Note: In the API source files, a TemplateAdaptor class is provided which

implements IMcuAdaptor. If you need to implement a new adaptor, we suggest to start with TemplateAdaptor.cpp and follow the TODO instructions inside. You are also invited to refer our [/examples/Arduino/] and [/examples/nRF52/] examples, in which the adaptor for Arduino and nRF52 are implemented.

ii) Data

We define three structs (which are placed in orange blocks in the picture) to encapsulate wavelength, spectrum, and XYZ data. Users can easily extract these structs from the return packet, and retrieve their interested information.

Example:

```
SpectrumInfo info;
nsp32.ExtractSpectrumInfo(&info);   // where "nsp32" is a NSP32 object
uint32_t count = info.NumOfPoints;
float *data = info.Spectrum;
```

Please refer [/examples/Arduino/] 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 such as CMD_ACQ_SPECTRUM, API handles the complete cycle (e.g., 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/Arduino/] examples to see how to use sync. and async. commands respectively with the API.

Customization

There are two areas users can do customization while using the API.

1) Customize a MCU adaptor
   If users are using Arduino or nRF52, the corresponding adaptor can just be found in our examples. Otherwise, as described in "Architecture & Concepts", users need to implement a specific adaptor for the users' selected MCU.
2) Customize a C wrapper
   This API is implemented in C++. So if users need to use it from a C source code (e.g. main.c), users might need a C wrapper (to wrap the API). Please refer the [/examples/nRF52/(SpectrumMeter/NanoLambdaNSP32)] example, which contains NSP32CWrapper.h and NSP32CWrapper.c. Then compile the NSP32CWrapper.c with C++, and compile the main.c with C (usually the file type settings can be changed within the IDE).

Development Tool Recommendation

IDE with C++ compiler

How to Use

1) Put your specific MCU adaptor files (e.g. FooAdaptor.h, FooAdaptor.cpp) into "NanoLambdaNSP32" folder.
2) Add "NanoLambdaNSP32" folder to your project.
3) In your IDE project settings, add "NanoLambdaNSP32" folder to the include paths.
4) Write codes (please refer our examples).
5) Build.

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 MCU, 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.

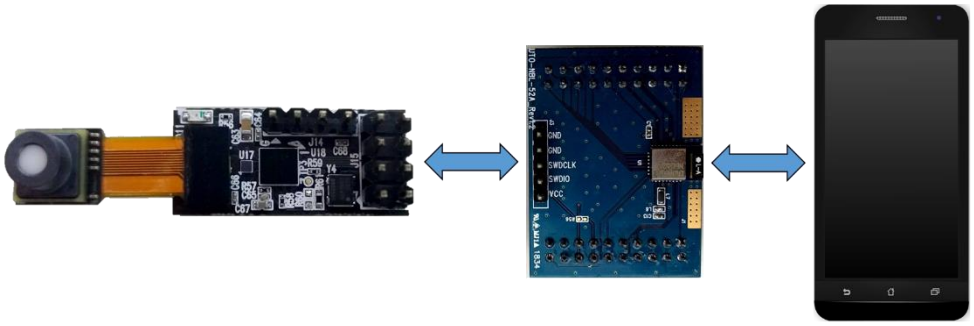
2) Auto wakeup: NSP32 module has two power modes: "active mode" and "standby mode". In standby mode, no commands would be accepted. If you are using our API for MCU, it auto checks NSP32 module's current mode before sending commands, and auto wakes it up if necessary.
   Note: When power consumption 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:

```
FooAdaptor adaptor();
NSP32 nsp32(&adaptor, NSP32::ChannelSpi);   // use SPI
```

   or

```
NSP32 nsp32(&adaptor, NSP32::ChannelUart);  // use UART
```

5) Both return packet raw bytes and easy-to-use structs are available: In most cases, users extract their desired information (struct data) from the return packet by calling NSP32::Extract...() functions. However users can also call NSP32::GetReturnPacketPtr() to access the return packet raw bytes if needed.
6) Support MCU as a forwarder: In the scenario MCU acts as a forwarder, see picture below for example, to forward command packets and return packets between the NSP32 module and a smart phone, so that the smart phone can wirelessly control NSP32 module and get the spectrum data. Our API provides a very simple way to perform this forwarding task by calling NSP32::FwdCmdByte() and NSP32::UpdateStatus(). Please refer [/examples/nRF52/] example for complete demonstration (see the main loop in the main() function).



NSP32 module MCU with BLE smart phone

Note: Users can replace the BLE with WIFI or other wired/wireless transmission protocols.

API Reference

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