Network Processor Interface User's Guide

Document Version: XXXX

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

References

[1] TI-RTOS User Guide

[2] TI BLE Vendor Specific HCI Reference Guide

[3] Z-Stack Monitor and Test (MT) API (need version still)

1 Overview

A common system configuration is to use the Texas Instruments CC26xx as a network processor (NP) where a host microcontroller (MCU) is able to control the CC26xx platform by sending serial commands. This requires certain power domains of the NP to be on in order to receive and handle these commands. It also requires translating serial packets into commands for the NP as well as translating responses into serial packets to be sent to the Host. The handling of these messages along with managing the power domains in an efficient manner is the responsibility of the network process interface (NPI) framework of the CC26xx platform. The NPI framework enables the CC26xx to go into low power modes when not being used and to be asynchronously woken up to respond to commands from the host MCU.

The purpose of this document is to give an overview of the Network Processor Interface framework to help with the development of software on the host MCU as well as aiding development of software on the network processor itself.

1.1 Introduction

The NPI framework is shared among the following (insert marketing name) products:

- Bluetooth Smart
- Z-Stack ZigBee Network Processor (ZNP)
- TIMAC MAC Network Processor (MACNP)
-

Between these different products, there are some differences in how data framed for serial communication, as well as how messages are handled within the NP. Despite these minor differences, shown in red, all products share the same basic NPI framework shown below:

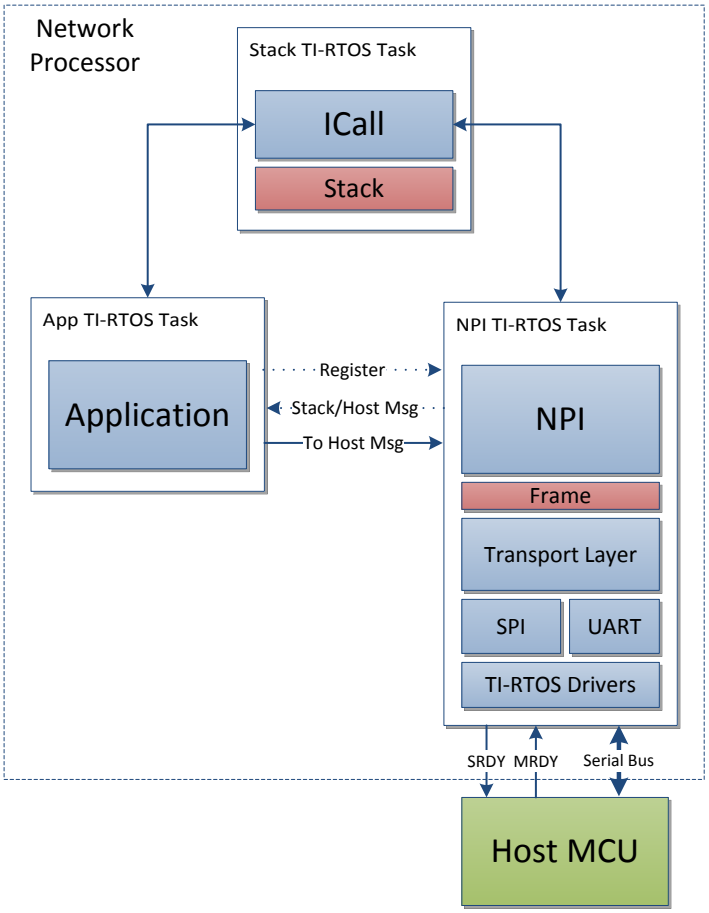

Figure 1 Network Processor Interface Framework

A more detailed description of the NPI sub-modules is included in (insert this section once it is written).

NPI's functionality can be broken down into three main components: managing the serial interface between the NP and Host, managing command and responses to and from the stack, and optionally routing stack or host messages to a registered application task.

1.2 Serial Interface

In order to properly manage the power domains of the CC26xx platform, NPI must include a way for an external host to wake the network processor from low power modes. This is done through the inclusion of "master ready", MRDY, and "slave ready", SRDY, signals. These two signals along with the respective SPI or UART serial bus signals comprise the serial interface of NPI.

MRDY is an input pin on the network processor and when set by the Host will wake the device enabling it to receive data over the serial bus. SRDY is an output pin on the network processor, and, when set by the NP, signals to the host that it may begin data transfer. This sequence of

setting MRDY and SRDY pins comprises a "handshake" process to guarantee both devices are awake and ready to send and/or receive data.

1.3 Command and Response Message Routing

The common use case of the NPI framework is a basic network processor. This means that the framework is strictly acting as a sending and receiving mechanism between the host MCU and the stack task. However, there also exist use cases that include some amount of "custom" application-dependent processing. For these cases not all messages from the host should be delivered to the stack and not all messages from the stack should be transferred directly to the host. To enable these types of applications, a mechanism to easily change the paths of inbound and outbound messages is built into NPI.

In Figure 1, the possible message rerouting with the application task can be seen. An application can easily interject messages to the host within the command and response traffic of the stack using the NPI APIs. An application can also register with NPI to either "Intercept" or "Echo" all messages originating from the host and/or all messages originating from the stack. This allows the application to modify messages as needed or to even process certain application specific messages without sending them to either the host or stack.

2 Interfacing with a Host Processor

The purpose of this section is to describe the expected behavior of host MCUs. This includes physical interface, NPI message types, timing of interface pins, and different message framing requirements of each software product. In order for a host MCU to properly interact with a CC26xx network processor, the below specifications should be followed.

2.1 Physical Interface

The NPI framework currently supports two serial interfaces. These are SPI and UART. The CC26xx has two SPI modules and only one UART module. When configured to use SPI, NPI will by default use the second SPI module (SSI1) (Note: This is due to the SmartRF06 board using SSI0 for driving LCD). For each serial interface, power management can be enabled and regardless of interface type MRDY and SRDY pins remain consistent. There are three possible configurations:

- UART with Power Management
- UART without Power Management
- SPI with Power Management

2.1.1 UART

NPI supports two different configurations of UART. One configuration only requires two pins, UART TX, and UART RX. This is an always on configuration that does not allow the CC26xx to enter into low power modes. The second configuration does allow low power modes because of the inclusion of the MRDY and SRDY pins for power management. Some UART interfaces include hardware flow control pins, CTS and RTS, however we do not currently support hardware flow control. Instead the MRDY and SRDY pins can be used for both power management and software flow control.

Table 1 NPI UART Pin Assignments

UART Signal Name	7x7: IOID	5x5: IOID	4x4: IOID	EM Header ID	SmartRF06 Port ID
RX	2	1	1	RF1.7	P408.14
TX	3	0	2	RF1.9	P408.12
MRDY	19	6	4	RF1.10	P403.12
SRDY	12	4	3	RF1.12	P403.16

2.1.2 SPI

There is only one SPI configuration which requires the use of MRDY and SRDY. The NPI framework requires the ability to asynchronously send data from NP to Host. Because of this requirement it is not possible to have a configuration with the traditional SPI signals alone (SCLK, MOSI, MISO, CS). The SRDY signal is used to signal the SPI master that a slave to master transmission is pending so the master should toggle the SPI SCLK signal to begin the transfer. Without this signal, an asynchronous NP to Host transfer is not possible. The MRDY signal performs the chip select, CS, function among other functions so the chip select signal is not needed with the SPI configuration.

NOTE: The signals highlighted in the table below suggest an alternative pin assignment for SPI using SSI1 to avoid a conflict with the LCD control signals on the SmartRF06.

Table 2 NPI SPI Pin Assignments

SPI Signal Name	7x7:IOID	5x5:IOID	4x4:IOID	EM Header ID	SmartRF06 Port ID
MOSI	9	11	9	RF1.18	P404.8
MISO	8	12	0	RF1.20	P404.10
SCLK	10	10	8	RF1.16	P404.4
MOSI	*23*	*n/a*	*n/a*	*RF2.5*	*P404.12*
MISO	*24*	*n/a*	*n/a*	*RF2.10*	*P404.18*
SCLK	*30*	*n/a*	*n/a*	*RF2.12*	*P405.2*
MRDY	19	6	4	RF1.10	P403.12
SRDY	12	4	3	RF1.12	P403.16

2.1.3 Configuration Management

With any network processor project that uses the NPI framework, switching between configurations can be easily managed. There are only two preprocessor defines that are needed.

- **POWER_SAVING** – If defined, MRDY/SRDY as well as low power modes are used. If not defined, the device will be always on and NPI must use UART.
- **NPI_USE_SPI/NPI_USE_UART** – Only one of these can be defined at a time. Each define corresponds to the underlying serial interface that will be used by NPI
- **NPI_SREQRSP** – If defined, support for synchronous REQ/RSP messaging is included. This is currently only relevant for MT-based NPI functionality.

2.2 NPI Messages

NPI *messages* describe an ordering of events on the interface pin, not the content of the data that is sent or received. NPI *frames* define data content that is sent or received. In this section, the focus will first be to describe messaging and the behavior of the physical interface and then to describe how data content, or frames, are then sent over these messages.

When using power management pins, MRDY and SRDY, there are only two events whose order determines the message: assertion of MRDY and the assertion of SRDY (Note: MRDY and SRDY are active low, thus "assertion" refers to a logic value of 0). Before any data can be sent, a "handshake" must occur and this handshake can be initiated by either Host or NP. There for two messages exist: Asynchronous Request – transfer initiated by Host, and Asynchronous Indication – transfer initiated by NP. When not using power management pins, the two events that define an Asynchronous Request or Asynchronous Indication is the beginning of a data transfer on the UART Master TX pin or the UART Slave TX pin respectively.

2.2.1 Asynchronous Request (AREQ)

An asynchronous request is a message that is sent from Host to NP. The host must initiate the handshake by asserting MRDY, once SRDY has been asserted, then data can be transferred

Host to NP. MRDY is de-asserted by the Host once all data has been transferred and then NP de-asserts SRDY notifying the Host that it may be in a low power mode. The order of events is shown below:

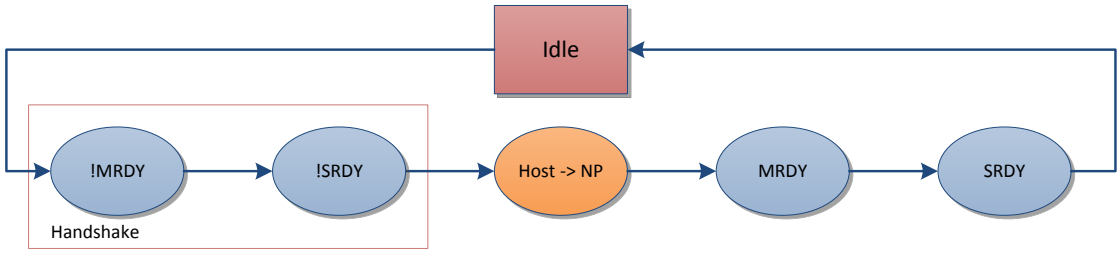

Figure 2 AREQ Order of Events

2.2.1.1 AREQ Timings

An AREQ message is presented in Figure 3. In this figure there is one AREQ message which ends once SRDY is de-asserted and a subsequent AREQ is initiated when MRDY is asserted. The following timings provided are based on testing. These do not represent absolute values but instead are values that were tested and shown to be functional.

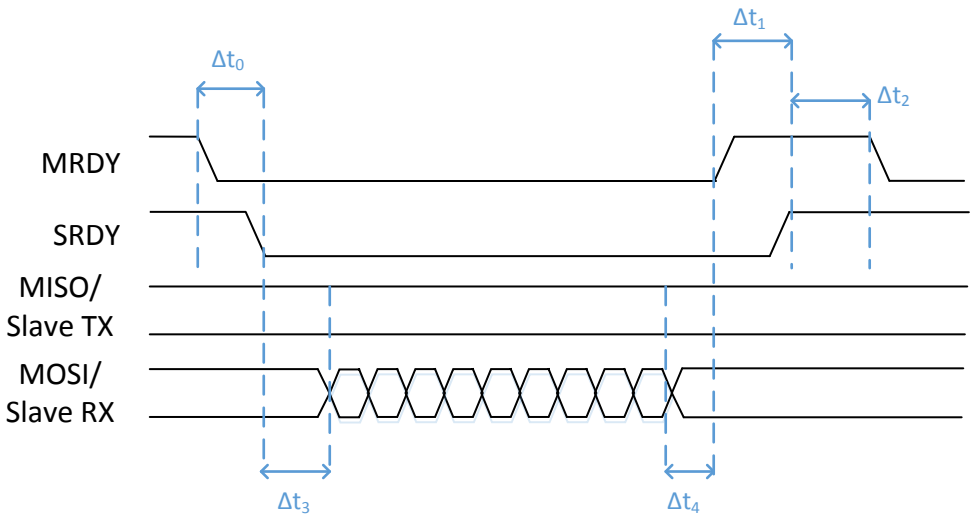

Figure 3 AREQ Timing Diagram

Table 3 AREQ Timings

Interval	Minimum (µs)	Average (µs)	Maximum (µs)
Δt_0	48	90	-
Δt_1	41	46	-
Δt_2	-	510	-
Δt_3	4.1	4.5	-
Δt_4	-	101	-

2.2.2 Asynchronous Indication (AIND)

An asynchronous indication is a message that is sent from NP to Host. The only difference between this message and AREQ is that SRDY is asserted first and the direction of the data transfer is inverted. The order of events can be seen below:

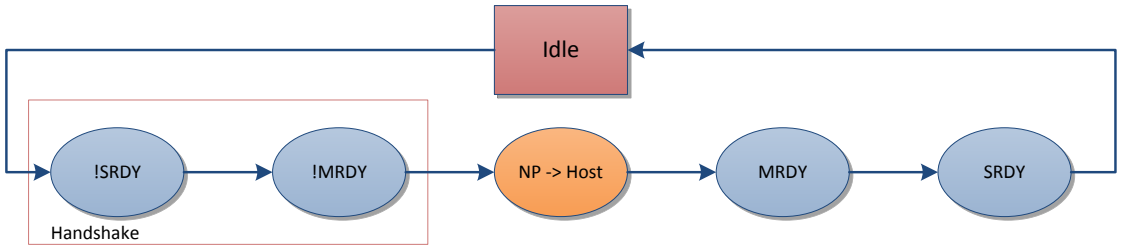

Figure 4 AIND Order of Events

2.2.2.1 AIND Timings

An AIND message is presented in Figure 3. In this figure there is one AIND message which ends once SRDY is de-asserted and a subsequent AIND is initiated when SRDY is asserted. The following timings provided are based on testing. These do not represent absolute values but instead are values that were tested and shown to be functional.

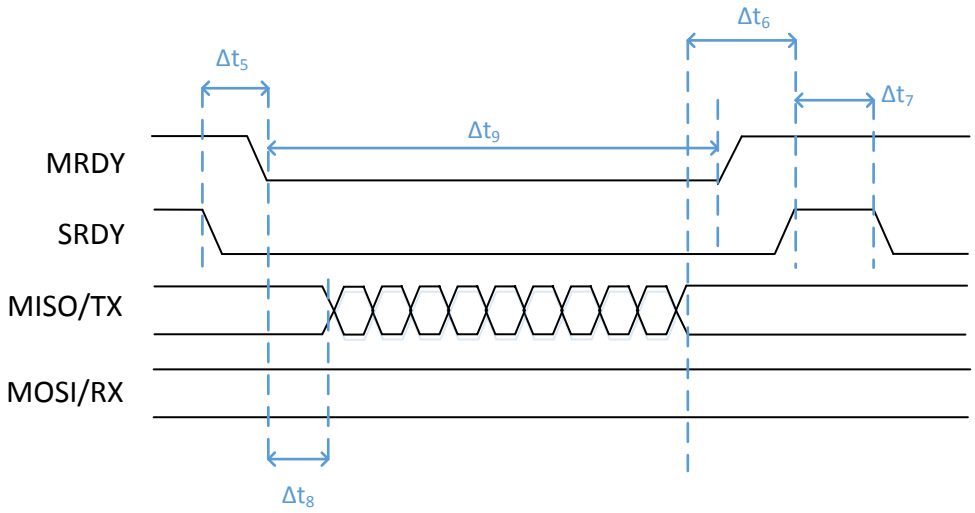

Figure 5 AIND Timing Diagram

Table 4 AIND Timings

Interval	Minimum (µs)	Average (µs)	Maximum (µs)
Δt_5	45	-	-
Δt_6	30	46	
Δt_7	270	450	-
Δt_8	.5	-	-
Δt_9	50	-	-

2.2.2.2 AIND Dependencies

The timing diagram in Figure 5 shows that MRDY is de-asserted only after the transfer on the serial bus has completed. This behavior however, is dependent on which underlying serial bus is being used.

For SPI, the NP has no concept of when the SPI transfer has finished because the Host is in control of SCLK and determines how many bytes are clocked. It is the Host's job then, to clock the correct amount of bytes and then de-assert MRDY which signals to the NP that the transfer has indeed completed. The correct amount of bytes is determined by a length field that is present with the SPI frame that is being transferred, which will be discussed further in Section **Error! Reference source not found.**.

For UART, the NP is in control of when the data is transferred and has knowledge of when it has completed a transfer to the Host. Thus, the host can de-assert MRDY prior to the actual end of the data transfer because NP is not waiting to be signaled. This reduces the amount of processing required to receive each message since the packet is not parsed during the actual transmission. The Host will know that is has received a full message once SRDY is de-asserted

2.2.3 Bidirectional Messaging

With any protocol that allows asynchronous messaging, where a signal transition denotes the start of a message as with MRDY and SRDY, there are inherent collisions between messages. Instead of requiring intricate collision handling, the NPI framework allows for bidirectional messaging to occur. This means that data can be sent from the Host to NP and from NP to the Host in the same message window regardless of the handshake order.

While reducing collision handling, bidirectional messaging adds some complexity to what operations must be performed or initiated by each device. For every AIND the NP initiates, it must prepare to read as well as write when MRDY is asserted. For every AREQ, the Host must prepare to read as well as write once SRDY is asserted. Each device will also need to handle any FIFOs that could potentially be overrun during a message and check at the end of every message to see what, if anything, has been received.

The scenario where bidirectional messaging will occur is when a device's input pin (SRDY wrt Host) has been asserted, and prior to asserting its output pin (MRDY wrt Host) the device gets a pending message to write. Below is the modified ordering of events where there exists now a partial order on the MRDY and SRDY assertions.

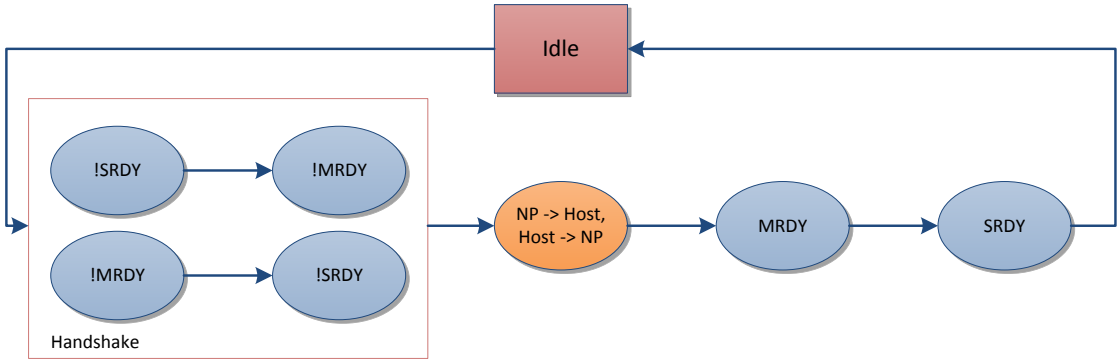

Figure 6 Bidirectional Messaging

As seen in Figure 6, it is still the responsibility of the Host to de-assert MRDY first. This again presents a serial bus dependent solution.

For SPI, MRDY can only be de-asserted after *both* enough bytes have been clocked out to transfer the message from Host to NP and the full message from NP to Host has been received. Again, the length of the message from NP can be found in the length field of the frame that is being sent.

For UART, MRDY can be de-asserted after the Host to NP transfer has completed. The NP will de-assert SRDY after MRDY has been de-asserted and the NP to Host transfer has been completed.

When power management pins are not used, the only allowed serial interface is UART. UART inherently is capable of handling bidirectional transfers because there are two dedicated and independent channels for sending data between Host and NP. In this situation, AIND and AREQ can be sent completely independently without the need for handshaking so the above techniques for handling handshaking collisions does not apply. Instead, AIND and AREQ messages should be sent and handled independently.

2.2.4 Fundamental Rules of MRDY and SRDY

With handshaking and bidirectional messaging, there are two basic rules to the operation of the serial bus:

1. **Each device must always initiate a read prior to asserting its respective output pin (MRDY wrt Host) regardless of the state of the its respective input pin (SRDY wrt Host)**
2. **Each device can only begin to write (or clock data in the case of SPI) once both MRDY and SRDY are asserted**

2.2.5 Synchronous Request/Response (SREQ/SRESP)

As mentioned previously, the NPI framework is common across multiple CC26xx software products. and protocol stacks. Some of these products, such as Z-Stack, require synchronous requests and responses.

A Synchronous Request (SREQ) is a *frame,* defined by data content instead of the ordering of events of the physical interface, which is sent from the Host to NP where the next frame sent from NP to Host must be the Synchronous Response (SRESP) to that SREQ. Because these are not messages but frames, they can be sent over AREQ and AIND messages. The NPI framework then does the proper handling to guarantee the next AIND message contains the SRESP frame, thus enforcing the synchronous behavior.

In the next section, NPI frames will be discussed in greater depth.

2.3 NPI Frames

An NPI frame defines the content of the data being sent. In order to assure reliable transmission of data over messaging, specific frame types are used for each underlying serial interface. This is done to help overcome some short comings of bidirectional communication for each.

Not only is there framing for NPI messages, there is also technology specific framing which defines the format of the data payload sent within NPI Frames. The following are examples of technology specific framing specifications:

- Z-Stack and TIMAC, MT (Monitor and Test)
- Bluetooth Smart, Host Command Interface (HCI)

In this section, the specific frame for each serial communication bus will be discussed as well as how technology specific frames can be sent over these NPI frames.

2.3.1 NPI SPI Frame

There are a few limitations of SPI that must be addressed by defining data fields of each NPI message. The first limitation is that the SPI master must trigger the clock (SCLK) signal in order to receive bytes from the slave. If the slave is sending an AIND message, then the master must know how long this message is in order to receive the complete message.

The other limitation of SPI is that if the clock signal is triggered, the master or slave must transmit empty bytes if they have nothing to send. The simple scenario when this occurs is during either an AIND or AREQ where either the master or slave receives only empty bytes. These messages of strictly empty bytes could be easily ignored but the bidirectional message scenario requires more complex handling. If the slave and master are both transmitting non-empty bytes then the shorter message will have to be padded with empty bytes so that the longer message can be fully transmitted. Determining which bytes of message bytes versus empty bytes in this scenario requires message delimitation.

In order to handle these limitations of SPI, the NPI SPI Frame is used for all NPI messages sent over SPI. The NPI SPI Frame has four fields: start of frame (SOF), length, data payload, and frame check sequence (FCS). The SPI Frame Format is shown in Figure 7.

Figure 7 NPI SPI Frame Format

The SOF byte always has the value of 0xFE. The length byte describes the length of the data payload field and the FCS byte is calculated from the length and payload field bytes. The FCS byte is calculated using the following where Len is the one byte length field and D_n is the n^{th} byte of the data payload:

$$FCS = Len \oplus D_0 \oplus \ldots \oplus D_n$$

This NPI SPI Frame handles both the delimitation of empty vs non empty data bytes as well as having a fixed length field allowing the SPI master to know how many times the SCLK signal must be toggled to receive a complete AIND message from the SPI slave.

2.3.2 NPI UART Frame

UART does not have the same inherent qualities as SPI. Thus, a much simpler frame format can be used. Since UART RX and TX channels are independent of each other there is no need for a predefined fixed length field and since there are no empty bytes that must be transmitted each frame does not need to be delimited. This allows one to treat every byte received over UART RX or TX channels as a valid byte of data payload thus the NPI UART Frame consists of only a data payload field. This very simple format can be seen in Figure 8.

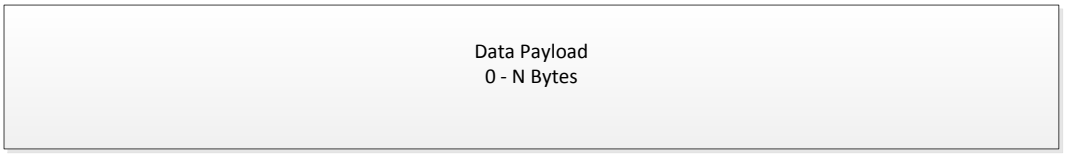

Figure 8 NPI UART Frame Format

2.3.3 Data Payload

All topics discussed to this point have been NPI Transport Layer specific implementation from the messaging to the frame format. These components are completely agnostic to what is being sent within the data payloads and simply create a stream of data bytes that are sent to the technology dependent frame parsers. As shown in Figure 1, the Frame module receives these bytes and interprets them according to a technology dependent specification so that these commands can be routed to the stack and/or application.

The fact that the NPI Transport Layer provides basically a byte stream of data from the Host and a byte stream to the Host allows for some flexibility with respect to how technology dependent commands or responses must be transferred. The main benefit or feature is that this allows for fragmentation of these commands or responses over the serial interface. For example, one HCI command does not need to be sent by the host over one message. In some cases, HCI or MT commands can even be larger than an NPI SPI frame so these must be sent as fragments.

For brevity's sake, the technology dependent messaging will not be covered in this document. However the following references will provide such information:

- Zigbee, MT: (Mike) [3]
- Bluetooth Smart, Host Command Interface (HCI): BLE Vendor Specific User's Guide [2]

3 Integrating NPI within an application

The intent of this section is provide information on how to integrate NPI within an already existing CC26xx (marketing name!) application. NPI can be used when implementing a custom network processor or as the back bone for a generic communication link with a host processor.

To begin, all NPI framework files are located within the following directories:

- Source: <install directory>/Components/npi
- Headers: <install directory>/Components/npi/inc

There are also a couple basic pre-processor defines which can be used to customize how the NPI interface behaves:

- POWER_SAVING – Allows the device to enter low power modes. If defined then MRDY/SRDY are used.
- NPI_USE_UART – Use UART for serial interface. Can be used with or without POWER_SAVING defined
- NPI_USE_SPI – Use SPI for serial interface. Can be used only with POWER_SAVING defined
- NPI_SREQRSP – Synchronous Response and Requests enabled
- NPI_TL_BUF_SIZE – The maximum size of an NPI message

There are more customizations possible, including changing which pins function as MRDY and SRDY, available in npi_config.h. Serial interface specific customizations, such as baud rate, are defined in serial interface specific files: npi_tl_uart.h/c or npi_tl_spi.h/c.

3.1.1 NPI Frame Modules and Creating a Custom Frame

Any project using NPI must include the <install directory>/Components/npi path. Almost all components in this NPI directory are common files that implement the transport layer or routing mechanism for messages. In Figure 9, these common components are shown in blue while the technology specific components are shown in red. These technology specific components are the frame parsers and constructors used to handle and prepare all data that will be sent over the NPI serial interface. These files are listed below. For all technologies there are also projects included in the SDK that take care of basic NPI integration into a network processor type application.

Table 5 Technology Specific Files

HCI	MT
<install dir>/inc/npi_ble.h	<install dir>/npi_frame_mt.c
<install dir>/npi_frame_hci.c	<install dir>/npi_client_mt.c
	<install dir>/npi_client.h

As mentioned previously, it is possible to implement a custom frame. Using a custom frame can decrease implementation overhead and greatly increase flexibility and adaptability of the NPI framework to any application. The technology specific files above are a good starting point to understand what is necessary to implement a custom frame. The next few paragraphs will discuss the APIs used and how to properly hook into the over NPI structure.

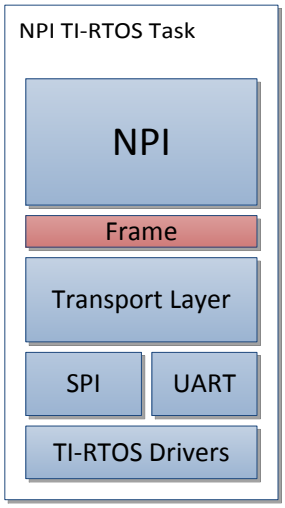

Figure 9 NPI Framework

There are two APIs that comprise most of a frame module. These are:

- NPIFrame_collectFrameData(void)
- NPIFrame_frameMsg(uint8 *pMsg)

NPIFrame_collectFrameData() is called after every NPI message is received. This function must parse through the received data and determine whether a complete frame has been collected. If a complete frame has been collected then that frame is translated into an NPIMSG_msg_t which is a specific type used for all messages routed through the NPI framework. The NPIMSG_msg_t consists of three fields: length, data, and type. Data is the frame that was collected and length is the length of that frame. Type can either by synchronous or asynchronous. If a frame is determined to be synchronous then no messages will be routed through NPI until the response to that synchronous message is sent back to the host. Asynchronous messages are routed in the order they are received. This function will invoke a call back registered by the NPI task indicated that a complete frame has been received and the NPI task then en-queues the message to be routed.

NPIFrame_frameMsg() receives a pointer to an array of raw bytes which must be translated into an NPIMSG_msg_t. This function is called any time there is a message that must be sent to the host. Again, if synchronous messages are used then determining the type of the message to be framed is important otherwise the asynchronous messages will be blocked awaiting a synchronous response to be sent to the host.

The basic implementation of a frame module can be seen in more detail in npi_frame_hci.c or npi_frame_mt.c. HCI only uses asynchronous messages while MT uses both synchronous and asynchronous messages. These files also give an example of how to implement the initialization function so that the NPI task can register to receive found frames.

3.1.2 Routing Data through Application

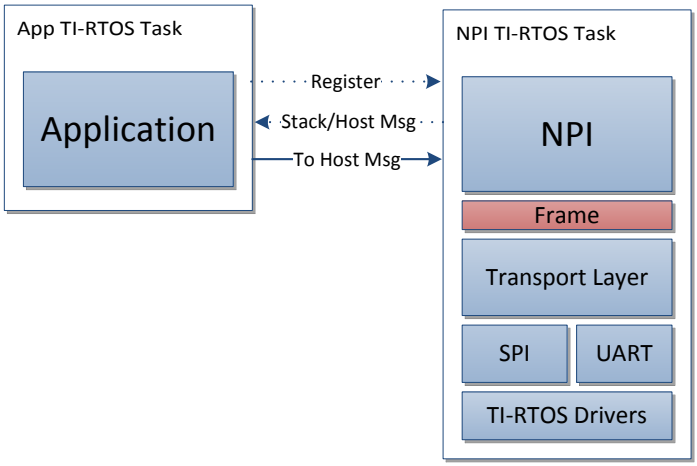

Figure 10 NPI and Application Interface

There are many cases where not all messages received from the host should be routed directly to the stack. There may be custom commands handled and required by the application. In order to handle such requirements, the NPI framework allows for an application to register for both incoming and outgoing messages.

There are two APIs provided to register for such messages:

- NPITask_registerIncomingRXEventAppCB()
- NPITask_registerIncomingTXEventAppCB()

Incoming RX events are messages that are being sent from the Host to NP while incoming TX events are messages that are being sent from the stack task to Host. The application registers by passing a function pointer to the above APIs. This function is then invoked passing the message as a parameter to this call back function. There are two different ways that messages are currently allowed to be routed. They can be routed as either ECHO or INTERCEPT. As ECHO, the application receives a copy of the message and another copy of the message is also sent to the stack. As INTERCEPT, the application will receive *all* messages and can choose to reroute them to the stack or to handle directly.

Registering for messages provides an NPI-to-Application routing of messages but there are also APIs which allow an application to send messages to the stack task or to Host.

- NPITask_sendToHost() – Send a message from Application to the Host
- NPITask_sendBufToStack() – Send a message from the Application to Stack task

These APIs to send messages to Host or Stack, and registering to receive incoming messages, give the application the complete flexibility to determine how all messages are handled and/or routed between the stack and the Host processor.