NEW RADIO COMMUNICATION FOR NEW ERA

NRC7394 Evaluation Kit User Guide

(Host Driver Porting)

Ultra-low power & Long-range Wi-Fi

Ver 1.0
Apr. 5, 2023

NEWRACOM, Inc.

NRC7394 Evaluation Kit User Guide (Host Driver Porting)
Ultra-low power & Long-range Wi-Fi

Office

NEWRACOM, Inc.
505 Technology Drive, Irvine, CA 92618 USA
http://www.NEWRACOM.com

Contents

List of Tables

List of Figures

1 Overview

This guide introduces the overall SW structure of NRC7394 host driver and gives some tips for applying the driver to other Linux hosts.

1.1 Software structure

As seen in Figure 1.1, NRC7394 host driver uses Linux Kernel features, SPI, GPIO, IRQ, mac80211, and netlink socket. SPI is used for I/O interface between the host, the NRC7394 module, and mac80211 for SW MAC, which is incorporated with MAC in the NRC7394 module. Netlink socket is used to communicate with user applications like CLI shell on host. GPIO including IRQ is used as an external interrupt source for flow control while communicating via SPI. The host applications hostapd, needs to be installed on the host to be operated as 11ah AP or wpa_supplicant as 11ah STA.

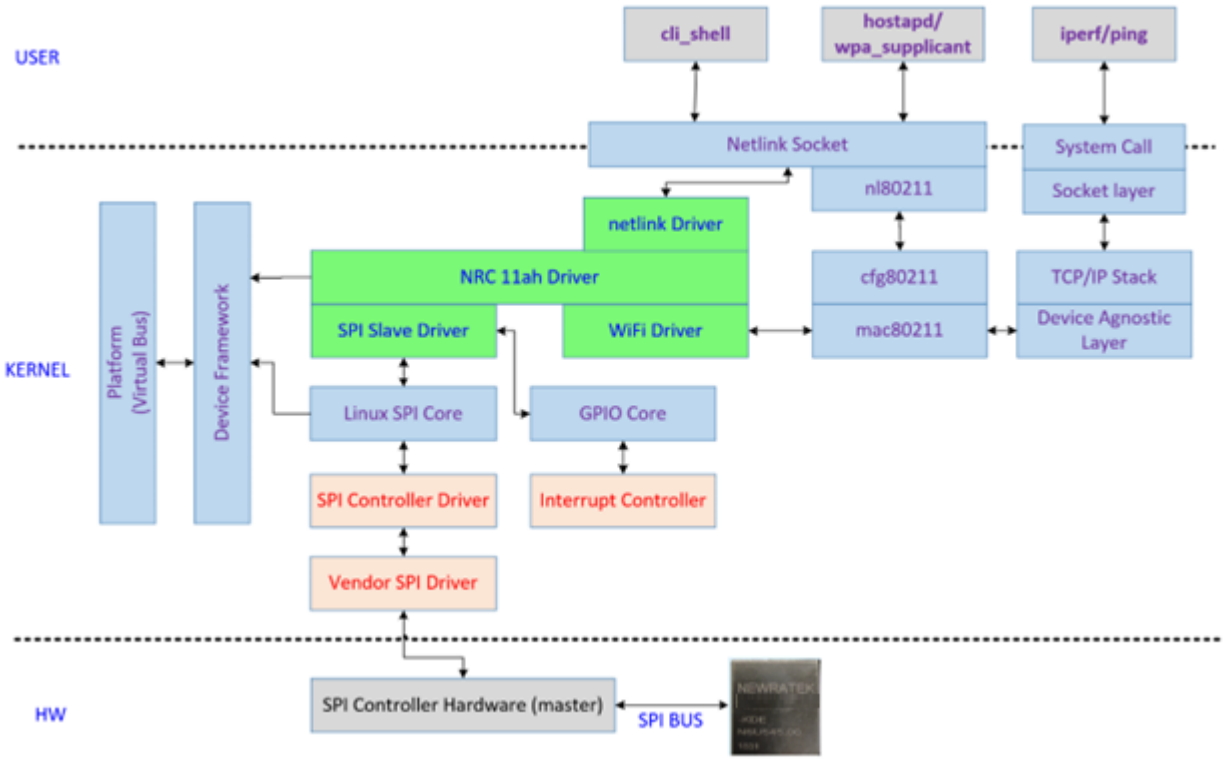

Figure 1.1 SW structure of NRC7394 host driver

1.2 Software components

NRC7394 host driver makes use of Linux Kernel APIs of GPIO, IRQ, SPI, and mac80211 so the host can support these components. To use Linux SPI/GPIO/IRQ framework or subsystem, each controller drivers provided by SoC vendor should be applied on host. If the framework is not supported on host, then the driver will be modified using other APIs (like vendor specific APIs). When it comes to mac80211, mac/cfg80211 modules should be built and loaded before loading NRC7394 host driver.

1.3 Hardware components

To communicate with NRC7394 module, SPI and GPIO (1 PIN) should be designated on Host. Figure 1.2 shows SPI and GPIO block on each Host (Raspberry Pi3) and NRC7394 Module. SPI is the main interface between Host, NRC7394 Module, GPIO, and used as external Interrupt source for flow control.

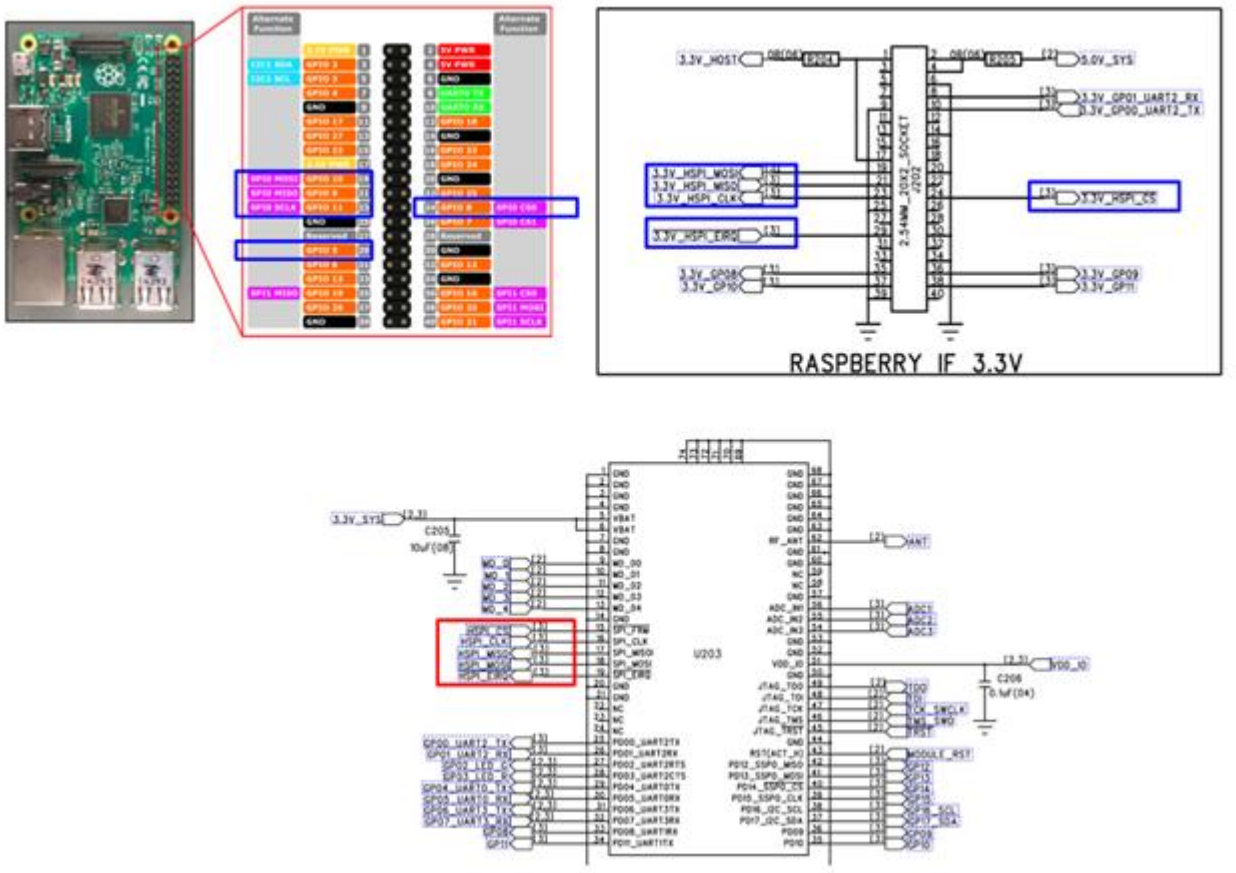

Figure 1.2 HW components on RP4 host and NRC7394 Module

2 How to build NRC7394 host driver

This chapters shows how to build NRC7394 host driver. There are 2 ways to build the driver: 1) direct-built on the host and 2) cross-compile on other hosts.

2.1 Direct compile

Before building NRC7394 host driver on host, Linux kernel sources or headers should be prepared on the host. Our drivers can cover Linux kernel version from 4.4.34 to 5.10.x.; to build properly, changing the kernel path in Makefile is needed.

2.2 Cross compile

It is a similar procedure as direct compile. But the only difference is the preparing the cross the compiler on your PC.

```
pi@raspberrypi:~/project/working/NRC_MACSW/host/linux/driver/nrc $ make

make[1]: Entering directory '/usr/src/linux-headers-4.14.70-v7+'
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-mac80211.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-trx.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-init.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-debug.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/hif.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/wim.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-fw.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-netlink.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-hif-cspi.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/mac80211-ext.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-stats.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-pm.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-dump.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-bd.o
  CC [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc-s1g.o
  LD [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc.mod.o
  LD [M]  /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.14.70-v7+'
```

Figure 2.1 Compile log

3 Source-code tree

Table 3.1 shows major files for NRC7394 host driver and description of them. If all the SW and HW components mentioned in Chap.1 is ready, then there is almost no need to modify source codes.

Table 3.1 Host driver files

	Category	Description
nrc-init.*	Module Initialization Linux Platform driver	Define module and module parameters Register platform device and driver
hif.*	HAL (HW Adaptation layer)	Wrapper functions for I/O
nrc-hif-cspi.*	HSPI driver	Functions for HSPI
nrc-mac80211.*	mac80211	Register driver to mac80211 Define mac80211 parameters
nrc-trx.*	Data Path	Define functions for data path
nrc-fw.*	Firmware	Define functions for FW download from host to target
nrc-netlink.*	Netlink socket	Define functions for netlink communication
nrc-pm.*	Power Management	Define functions for Wi-Fi Power Management
nrc-debug.*	Debug	Define functions for debugging
mac80211-ext.*	Utility function	Utility function about ieee80211
nrc-dump.*	Debug	Save Core Dump file when F/W Asserted
nrc-pm.*	Power Save	Function for Power Save
nrc-stats.*	Statistics	Function for Statistics to check a SNR/RSSI
wim.*	WIM	Function for handling the information message between host and firmware
nrc-vendor.h	Vender IE	Definition constant/type for Vender IE
fastboot-cm0.h	Boot	Second boot loader for CM0
nrc-build-config.h	Build	Define a definition/constant for building and configuration for host driver
nrc-wim-types.h	WIM	Define a data type for the WIM
nrc-rpi.dts	Device Tree	DT Overlay for RPI

3.1 Module Parameters

As seen in Figure 1.1, NRC7394 host driver support module parameters.

```
pi@raspberrypi:~/project/working/NRC_MACSW/host/linux/driver/nrc $ modinfo nrc.ko
filename:       /home/pi/project/working/NRC_MACSW/host/linux/driver/nrc/nrc.ko
description:    Newracom 802.11 driver
license:        Dual BSD/GPL
author:         Newracom, Inc.(http://www.newracom.com)
srcversion:     5D973059FE6887F20C9D358
alias:          spi:nrc80211
depends:        mac80211,cfg80211
name:           nrc
vermagic:       4.14.70-v7+ SMP mod_unload modversions ARMv7 p2v8
parm:           fw_name:Firmware file name (charp)
parm:           bd_name:Board Data file name (charp)
parm:           hifport:HIF port device name (charp)
parm:           hifspeed:HIF port speed (int)
parm:           spi_bus_num:SPI controller bus number (int)
parm:           spi_cs_num:SPI chip select number (int)
parm:           spi_gpio_irq:SPI gpio irq (int)
parm:           spi_polling_interval:SPI polling interval (msec) (int)
parm:           spi_gdma_irq:SPI gdma irq (int)
parm:           loopback:HIF loopback (bool)
parm:           lb_count:HIF loopback Buffer count (int)
parm:           disable_cqm:Disable CQM (0: enable, 1: disable) (int)
parm:           listen_interval:Listen Interval (int)
parm:           bss_max_idle:BSS Max Idle (int)
parm:           bss_max_idle_usf_format:BSS Max Idle specified in units of usf (bool)
parm:           enable_short_bi:Enable Short Beacon Interval (bool)
parm:           enable_legacy_ack:Enable Legacy ACK mode (bool)
parm:           enable_beacon_bypass:Enable Beacon Bypass (bool)
parm:           enable_monitor:Enable Monitor (bool)
parm:           bss_max_idle_offset:BSS Max Idle Offset (int)
parm:           macaddr:MAC Address (charp)
parm:           power_save:power save (int)
parm:           sleep_duration:deepsleep duration of non-TIM mode power save (array of int)
parm:           wlantest:wlantest (bool)
parm:           ndp_preq:Enable NDP Probe Request (bool)
parm:           ndp_ack_1m:Enable 1M NDP ACK (bool)
parm:           enable_hspi_init:Enable HSPI Initialization (bool)
parm:           nullfunc_enable:Enable null func on mac80211 (bool)
parm:           auto_ba:Enable auto ba session setup on connection / QoS data Tx (bool)
parm:           sw_enc:Use SW Encryption instead of HW Encryption (int)
parm:           signal_monitor:Enable SIGNAL(RSSI/SNR) Monitor (bool)
parm:           enable_usn:Use configuration of KR USN (Same ac between data and beacon) (bool)
parm:           debug_level_all:Driver debug level all (bool)
parm:           credit_ac_be:(Test only) credit number for AC_BE (int)
parm:           discard_deauth:(Test only) discard TX deauth for Multi-STA test (bool)
parm:           dbg_flow_control:Print slot and credit status (bool)
parm:           bitmap_encoding:(NRC7292 only) Use bitmap encoding for block ack (bool)
parm:           reverse_scrambler:(NRC7292 only) Apply scrambler reversely (bool)
parm:           power_save_gpio:gpio for power save (array of int)
parm:           beacon_loss_count:Number of beacon intervals before we decide beacon was lost (int)
```

Figure 3.1 Module Parameters

According to HW configuration, some SPI and GPIO parameters like chip_select, bus_number, max_speed_hz, etc. should be changed when driver is inserted to kernel. For SPI, user can set using spi_bus_num, spi_cs_num, spi_gpio_irq and hifspeed in module parameters.

4 Host SPI

NRC7394 contains SPI slave engine for a host interface. The SPI slave engine consists of two separate domain, device, and host side, as shown in Figure 4.1. This separation is mainly for power save. In a deep-sleep mode, most parts of NRC7394 including device side are turned off, but the host side keeps awake to monitor wake-up trigger from the external host.

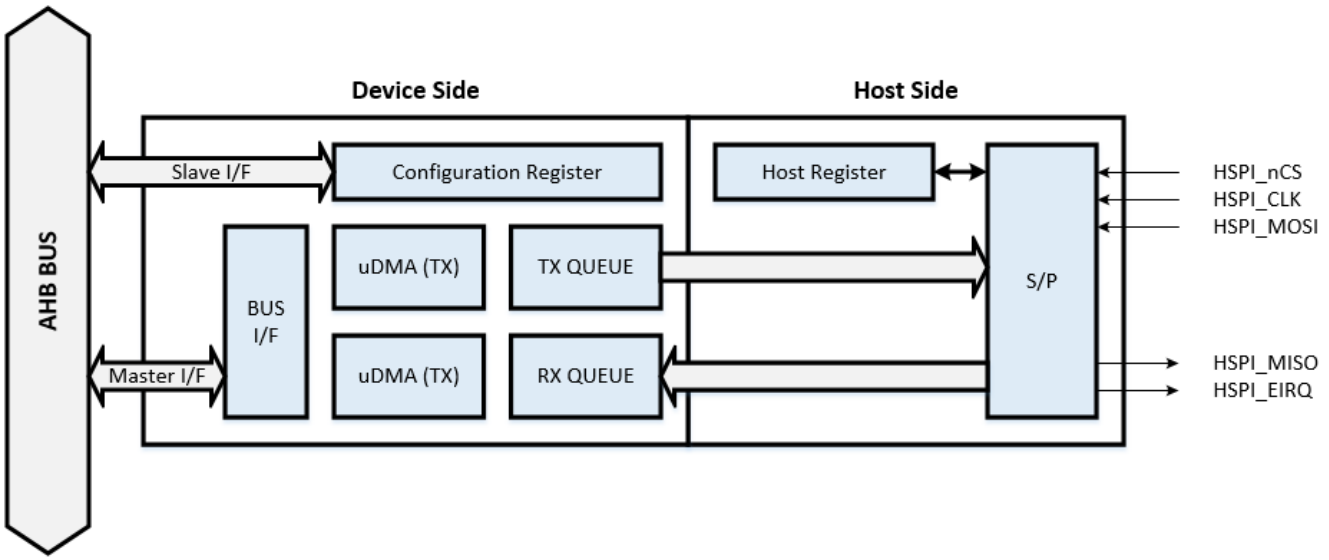

Figure 4.1 Block diagram of SPI slave engine of NRC7394

Table 4.1 HSPI pin description

Pin Name	I/O	Description
HSPI_EIRQ	Output	Interrupt to external host
HSPI_MOSI	Input	Master out Slave in
HSPI_MISO	Output	Master in Slave out
HSPI_nCS	Input	Chip select (active low)
HSPI_CLK	Input	Clock

A total of 5 dedicated pins are assigned for HSPI interface as presented in Table 4.1.

To use HSPI interface, the BOOT mode must be configured to ROM BOOT mode. The ROM BOOT mode can be selected by tying “GP19/Mode” pin to ground. The VDD_IO is the power supply for HSPI IOs.

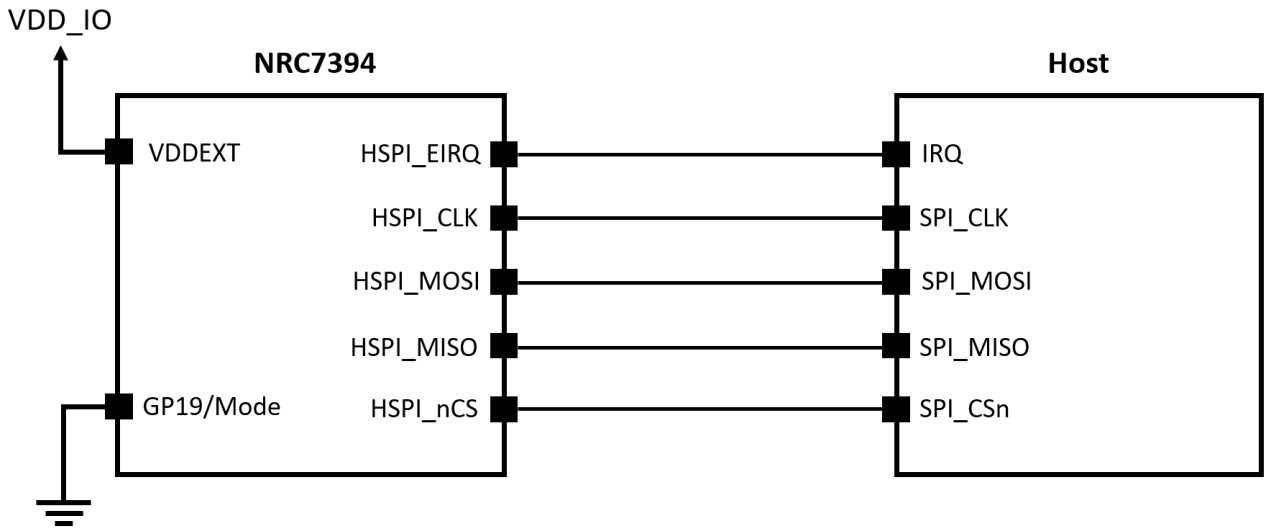

Figure 4.2 An example of H/W configuration for HSPI interface

SPI slave engine asserts HSPI_EIRQ interrupt when its TX QUEUE has data to send, or the status of RX QUEUE is changed. Therefore, when the HSPI_EIRQ is asserted, the external host needs to check what triggers the interrupt after clearing the interrupt by reading EIRQ_CLEAR register(0x12).

As shown in Figure 4.3, SPI slave engine supports SPI mode 0 (CPOL = 0 & CPHA = 0). The HSPI_nCS must be held low for entire read/write cycle and must be taken high at least one clock period after the read/write cycle completed.

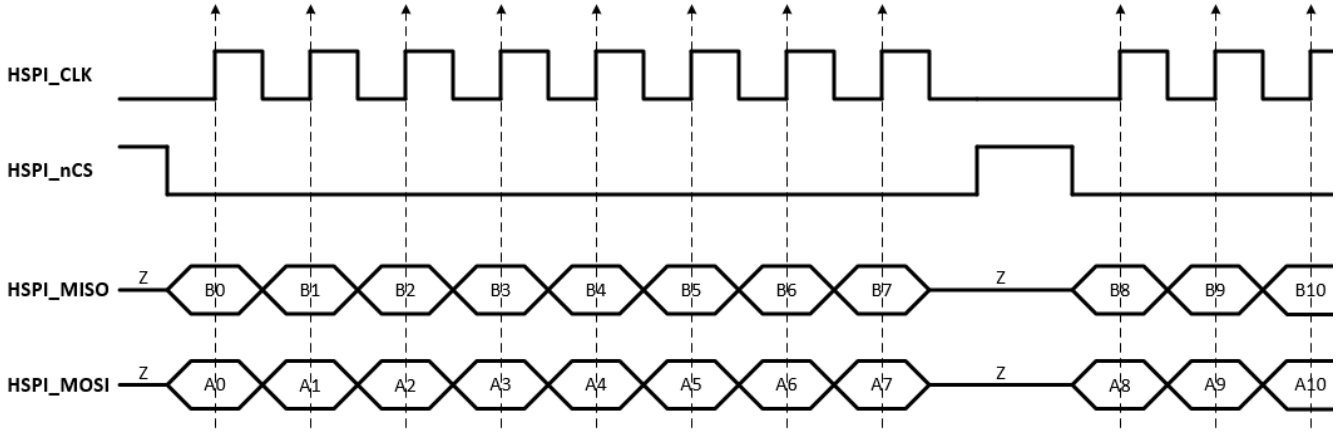

Figure 4.3 Timing diagram of HSPI interface

4.1 Single transfer mode

The single and burst transfer modes are defined. In single transfer mode, only one byte of data can be transferred from host to slave and vice versa. Figure 4.4 and Figure 4.5 represent the frame format for HSPI master write and read, respectively in single transfer mode. The single transfer mode starts with the command period and ends with the response period. The command period is composed of a 32-bits argument and a 16-bits cyclic redundancy check (CRC) part. The host (HSPI master) should check the acknowledgment (ACK) before sending the next command when writing data to HSPI slave. The ACK in response period is sent from HSPI slave to HSPI master to inform that it receives command correctly. When reading data, the HSPI slave transmit the data in response period.

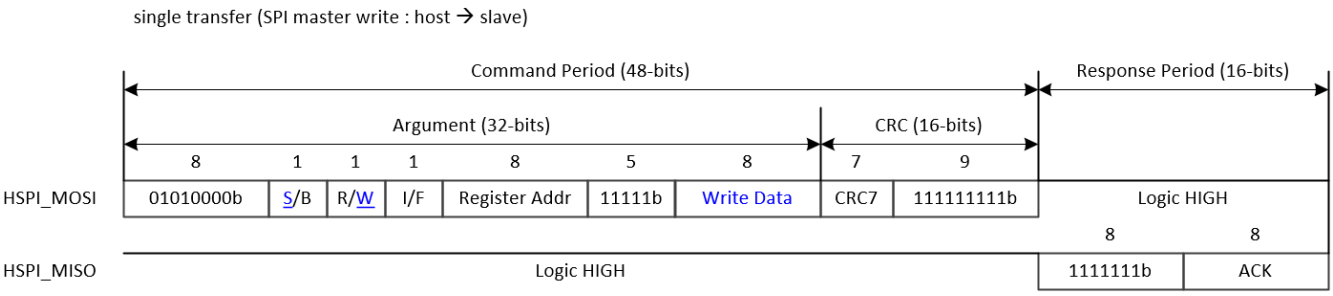

Figure 4.4 Frame format of HSPI master write in single transfer mode

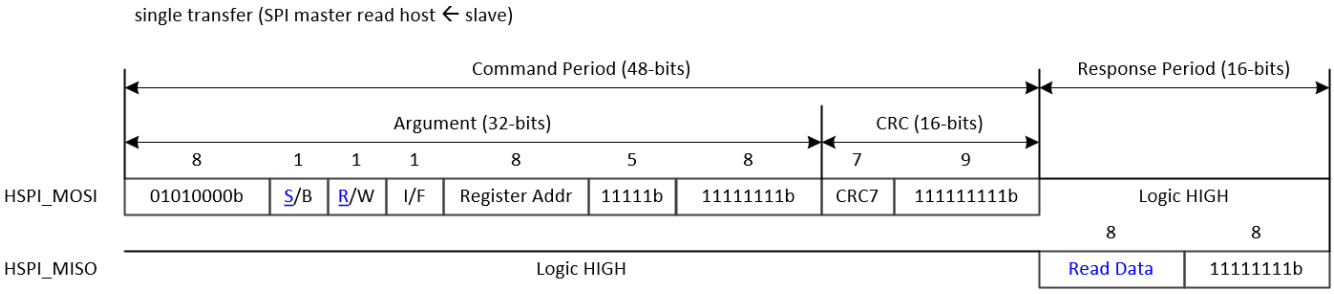

Figure 4.5 Frame format of HSPI master read in single transfer mode

Table 4.2 Field description of frame in single transfer mode

Field		# bits	Description
Argument	01010000b	8	header of the command period
	S/B	1	0 : single transfer 1 : burst transfer
	R/W	1	0 : read 1 : write
	I/F	1	0 : address increment 1 : address fix
	Register Addr	8	register address to access
	11111b	5	stuff bits
	Write Data / Stuff bits	8	when R/W = 0 : 0xFF when R/W = 1 : write data
CRC	CRC7	7	7 bits CRC calculation based on 32-bits argument
	111111111b	9	stuff bits
Response	Read Data / Stuff bits	8	when R/W = 0 : read data @ register address when R/W = 1 : 0xFF
	ACK	8	0x47

4.2 Burst transfer mode

Burst transfer mode also starts with the command period followed by a response period like the single transfer mode. However, additional data period follows the response period as shown in Figure 4.6 and Figure 4.7. In the data period, HSPI master writes or reads a number of data.

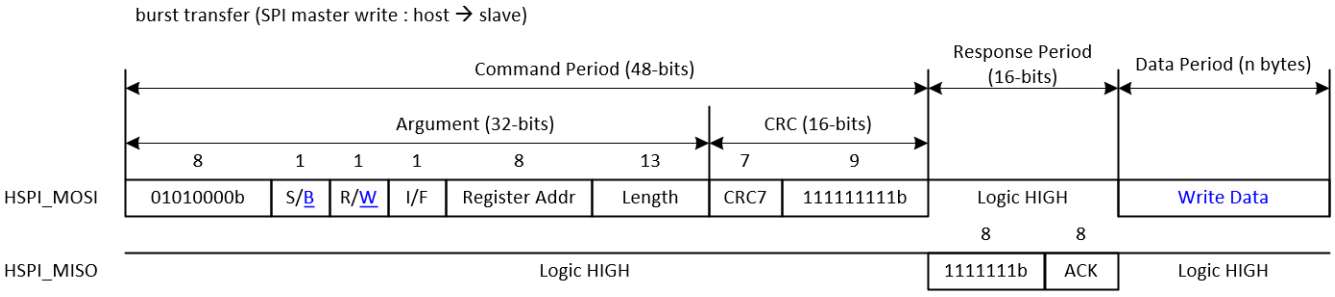

Figure 4.6 Frame format of HSPI master write in burst transfer mode

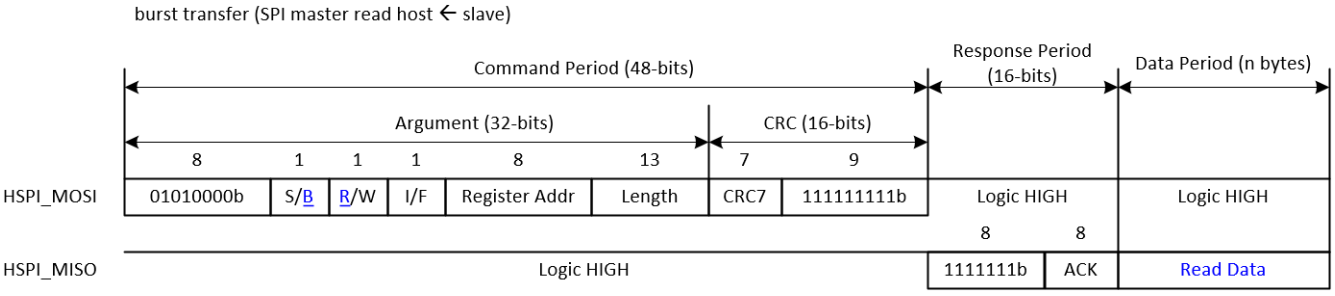

Figure 4.7 Frame format of HSPI master read in burst transfer mode

The frame structure of burst transfer mode is almost the same as the structure of the single transfer mode except for the length field at the end of the 32-bit argument. This 13-bit length field can represent up to 8K bytes and indicates the data size in byte.

Table 4.3 Field description of frame in burst transfer mode

Field		# bits	Description
Argument	01010000b	8	header of the command period
	S/B	1	0 : single transfer 1 : burst transfer
	R/W	1	0 : read 1 : write
	I/F	1	0 : address increment 1 : address fix
	Register Addr	8	register address to access
	Length	13	data length in byte
CRC	CRC7	7	7 bits CRC calculation based on 32-bits argument
	111111111b	9	stuff bits
Response	Read Data / Stuff bits	8	when R/W = 0 : read data @ register address when R/W = 1 : 0xFF
	ACK	8	0x47

5 Host Interface Layer

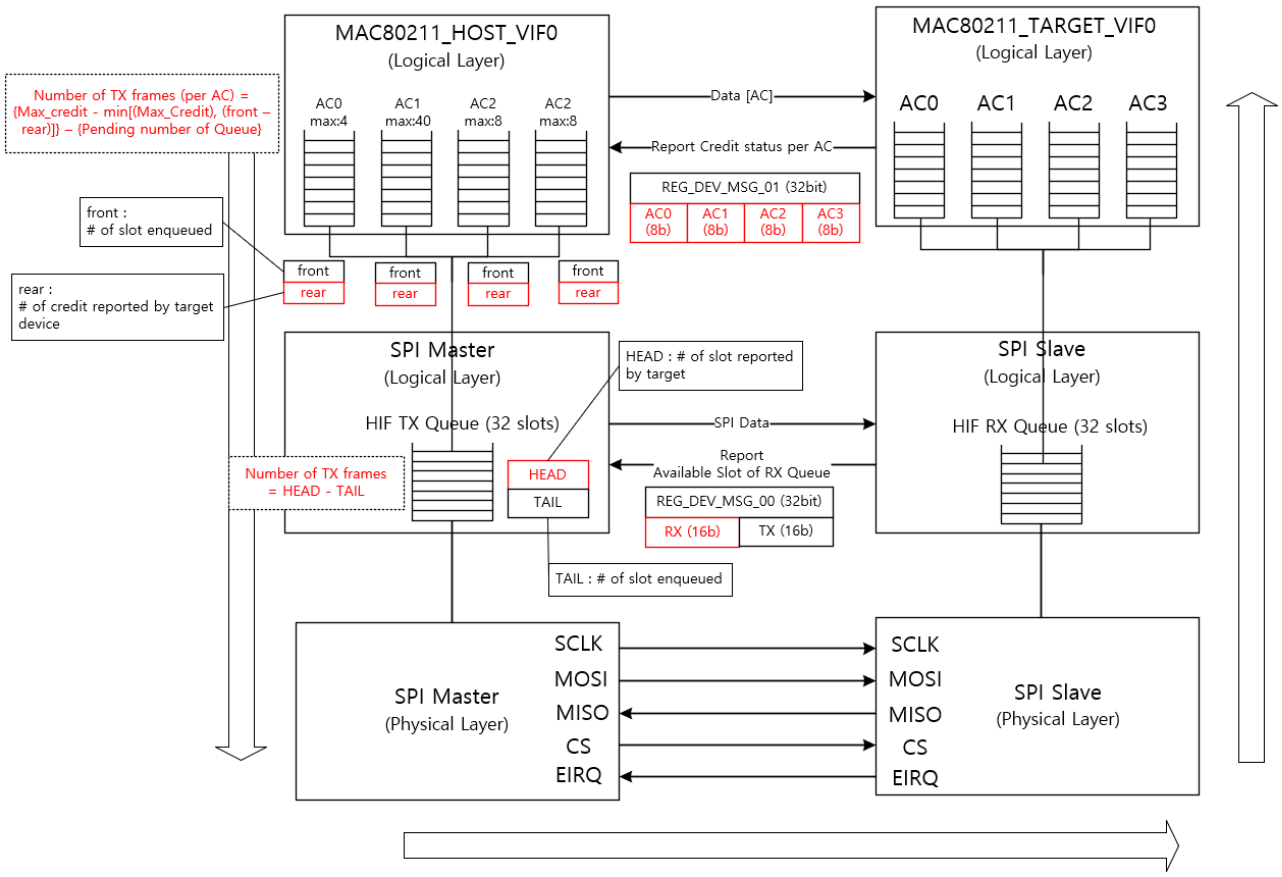

Figure 5.1 Slot and Credit (TX path on Host vs RX path on Target Device)

HIF (Host interface) layer located between MAC80211 (Linux Provide) and CSPI driver (Newracom Provide) is responsible for TRX data path. Basically, it abstracts various physical layers such as HSPI, SPI, UART, and SDIO, but we only focus on HSPI in this document.

HIF layer attaches its own header (called HIF Header) to payloads that are delivered from the upper layer (MAC80211), and then transmits them to target device via HSPI. On the contrary, HIF layer extracts some fields from HIF header generated by FW on target device whenever receiving frames from target device, and then finally delivers them to upper layer (MAC80211) without HIF header. (i.e. only MAC payload) The detailed structure of the HIF header will be described later.

As mentioned in previous chapter, there is “flow control function” that smoothly controls influx and outflux of frames between host and target device by SW. For this, two concepts are introduced, one is “slot” and the other is “credit”. ‘Slot’ is for preventing buffer overflow on target device and ‘Credit’ is for prioritization of AC (Access Category) used by 802.11 QoS. Credit is applied only for TX (not RX). Target device conditionally and repeatedly reports its TRX buffer status to host by asserting HSPI_EIRQ,

and then host should read the status before enqueueing frames on its TX queue or reading frames via HSPI.

Figure 5.1 shows this concept of slot and credit for TX Path. There are three independent layers, SPI physical layer, SPI logical layer (with slot), MAC80211 logical layer (with credit). HSPI_EIRQ that is asserted by target device makes host read the status register that informs available RX slot number on target device. Host has to decide whether or/and how many frames to transmit to target according to this slot report (refer to HEAD) and its own TX queue status (refer to TAIL). For example, there are 10 frames already enqueued in TX queue on host and host was reported with 15 available RX slots from target device. Then host can transmit only 5 frames via HSPI. In this condition, if host transmits 7 frames via HSPI, there might be overflow of RX queue on target device. The concept of credit is almost the same except that it is per AC (Access Category) and VIF (Virtual Interface). Similarly, according to credit report from target device and its TX queue status per AC (and VIF), host decides whether or/and how many frames are dequeued from each AC's queue and enqueued to HIF TX queue. However, this calculation is NOT done just by front (enqueued number of frame) and rear (reported credit number) like "rear – front". "Max Credit number per AC (AC0:4, AC1:40, AC2:8, AC3:8)" is introduced for enough influx frames to HIF TX queue. It is because handling frames on HIF TX Queue is done in the context of "workqueue" (refer to Linux workqueue concept). So credit calculation is finally done by "{Max_Credit_per_AC – min[Max_Credit, (front- rear)]} – {Pending Number of Queue per AC} " considering prioritization of each AC.

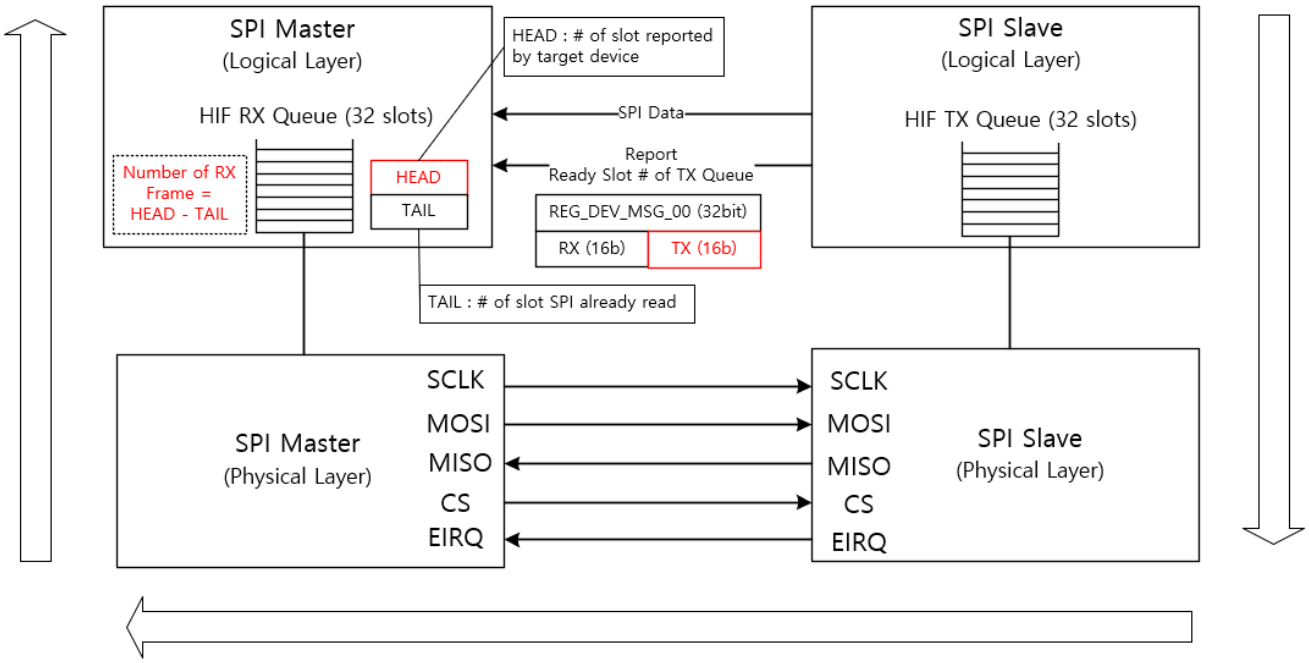

Figure 5.2 Slot and Credit (RX path on Host vs TX path on Target Device)

Figure 5.2 shows slot concept for RX path on host. It's much simpler than TX path because there is no credit concept. Host decides whether or/and how many frames to read via HSPI using TX Ready Slot Report from target device (refer to HEAD) and total number of frames read via HSPI (refer to TAIL). For

example, host is reported with 15 TX Ready slots from target device, and then host starts receiving frames until it reads 15 frames via HSPI.

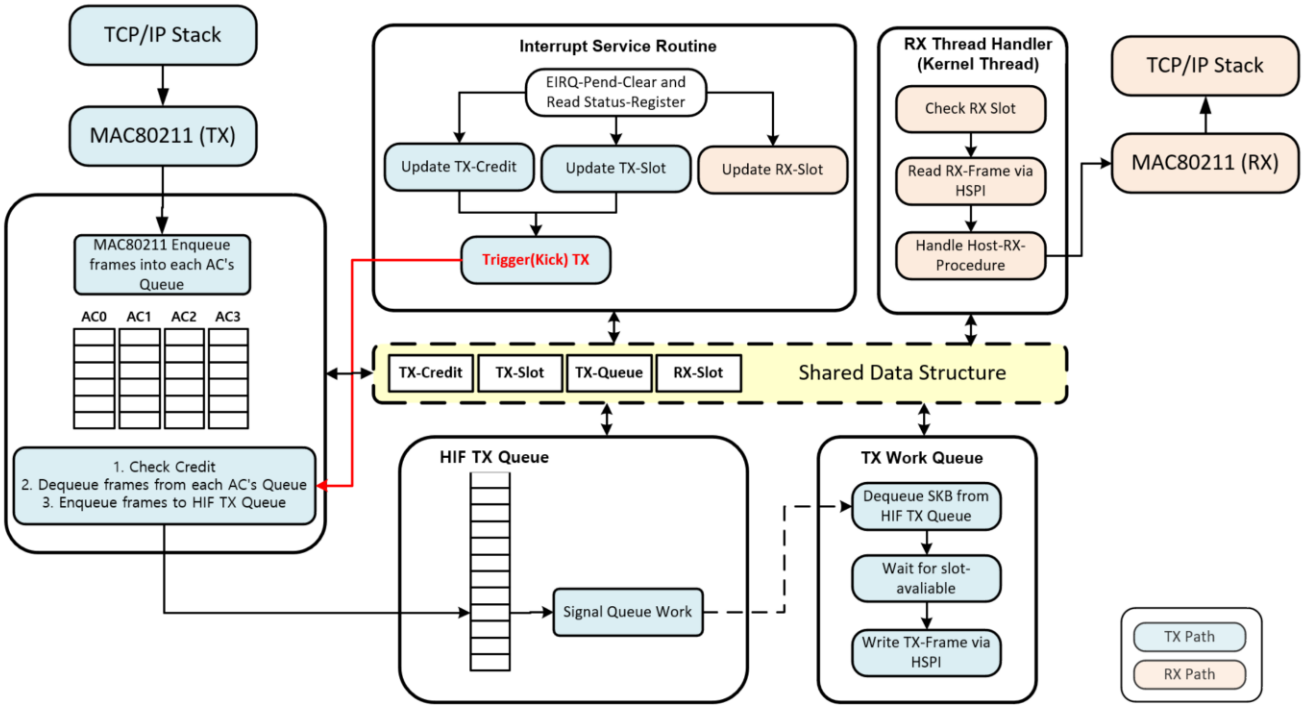

Figure 5.3 Host Interface Layer (TRX operation for ISR)

Figure 5.3 shows HIF Layer in general. For resource limitation on NRC7394 SoC, HSPI slave hardware engine has very small number TX/RX QUEUE (FIFO) inside, so TX credit and TRX slot reported using EIRQ and related registers are crucial. The condition of asserting EIRQ by target device is explained in the next section.

5.1 Interrupt Handling

Figure 5.3 shows the TRX operation with interrupt handling. Once the HSPI_EIRQ is asserted, EIRQ ISR (Interrupt Service Routine) checks what triggers interrupt after clearing interrupt by reading EIRQ_CLEAR (0x12) register. If the interrupt is caused by the “Credit Report or RX-Slot Report”, host kicks TX operation. If target device asserts EIRQ for Report TX-Slot, host stats RX operation.

NRC7394 (Target Device) asserts HSPI_EIRQ when the following events are occurred.

- TX credit updated
- TX QUEUE status updated
- RX QUEUE status updated

6 Wireless Information Message (WIM)

The WIM is used to request the operation of the firmware on target device. For example, Host Driver requests to set the frequency and BSSID after successful association. In the contrast, WIM is also used to notify host of some events occurred by firmware on target device.

The WIM protocol works base on HIF protocol structure as below.

HIF-Header(8B)	WIM-Header(4B)	TLVs(n byte)

```
struct hif_hdr {
    uint8_t type;
    uint8_t subtype;
    uint8_t flags;
    int8_t vifindex;
    uint16_t len;
    uint16_t tlv_len;
} __packed;
```

```
struct wim_hdr {
    union {
        uint16_t cmd;
        uint16_t resp;
        uint16_t event;
    };
    uint8_t seq_no;
    uint8_t n_tlvs;
} __packed;
```

HIF header indicates payload type such as Frame (data, mgnt, control), WIM, etc.

HIF_TYPE_WIM in hif_hdr should be set if the frame type is the WIM. The struct wim_hdr indicates how many Type-Length-Value (TLV) were loaded in the payload like frequency(one TLV), BSSID(one TLV), etc.

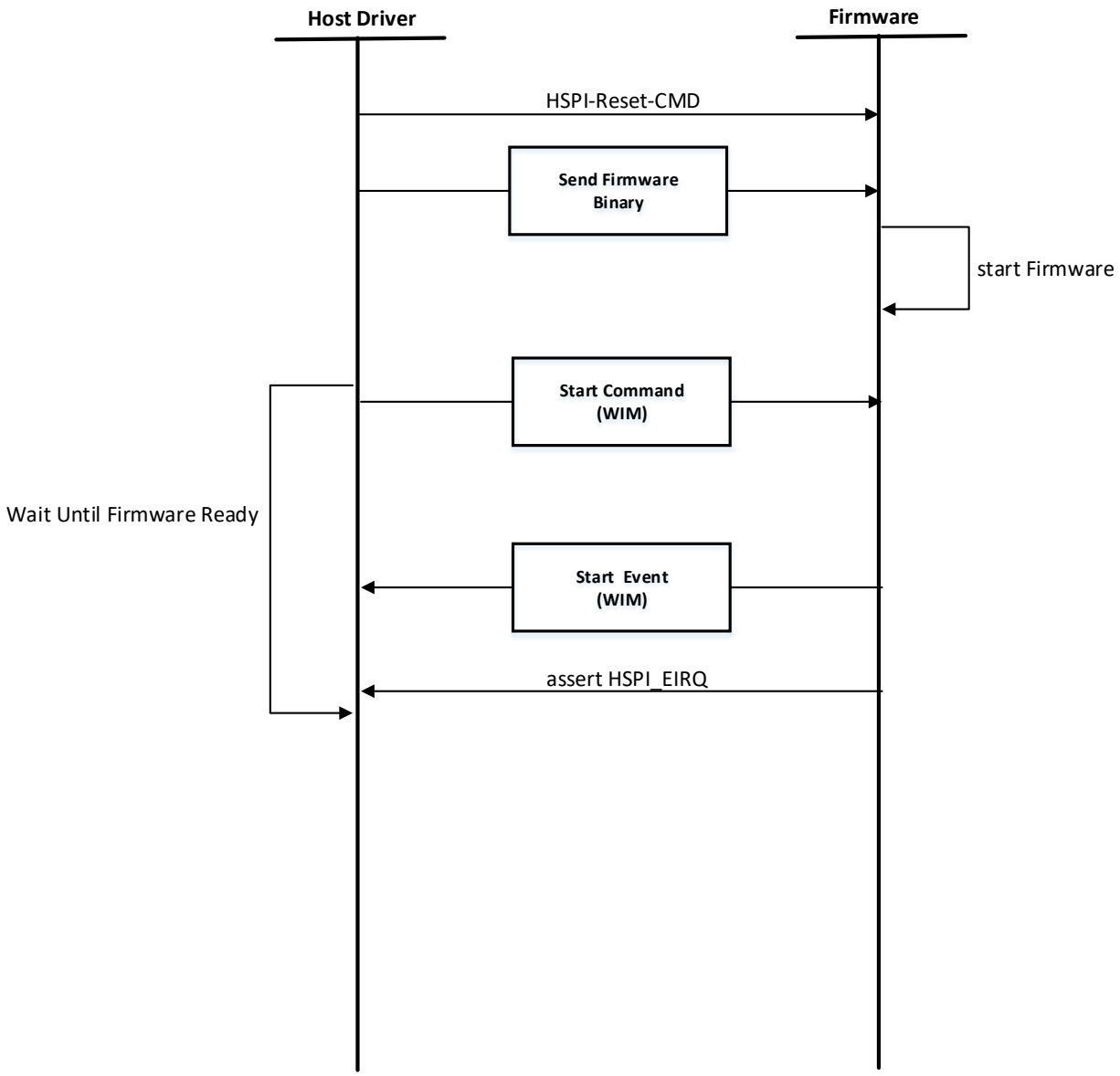

Figure 6.1. Initialize Sequence

Figure 6.1 describes how host driver downloads the firmware via HSPI and initializes it with WIM command and WIM event.

Host must wait until WIM Event comes back after issuing Start command.

6.1 WIM Command

This document explains basic WIM in the source code.

Table 6.1 WIM command

NAME	NOTE
WIM_CMD_START	Start command for firmware. Wait for start event from firmware
WIM_CMD_STOP	Start command for firmware when it unloaded the host driver
WIM_CMD_SCAN_START	Start scan command.
WIM_CMD_SCAN_STOP	Stop scan command
WIM_CMD_SET_KEY	Security key set command
WIM_CMD_DISABLE_KEY	Security key remove command
WIM_CMD_STA_CMD	Station information update command
WIM_CMD_SET	Set various TLVs command
WIM_CMD_REQ_FW	Firmware binary update command

6.2 WIM Event

Table 6.2 WIM event

NAME	NOTE
WIM_EVENT_SCAN_COMPLETED	Event to host when firmware completes the scan
WIM_EVENT_READY	Event to host when firmware completes initialization
WIM_EVENT_CREDIT_REPORT	Event to host when the credit has changed in firmware.

6.3 WIM TLVs

Table 6.3 WIM TLVs

NAME	NOTE
WIM_TLV_BSSID	uint8_t bssid[6] set bssid to firmware
WIM_TLV_MACADDR	uint8_t macaddr[6] set mac address to firmware
WIM_TLV_AID	uint16_t aid set aid to firmware
WIM_TLV_STA_TYPE	uint32_t sta_type set station type to firmware [WIM_STA_TYPE_STA, WIM_STA_TYPE_AP]
WIM_TLV_SCAN_PARAM	struct wim_scan_param set scan parameter for firmware scaning

WIM_TLV_KEY_PARAM	struct wim_key_param set security parameter for firmware security engine
WIM_TLV_STA_PARAM	struct wim_sta_param set sta parameter for firmware station information
WIM_TLV_READY	struct wim_ready_param various firmware parameter for host driver
WIM_TLV_AC_CREDIT_REPORT	struct wim_credit_report_param credit report for host driver from firmware
WIM_TLV_CH_BW	struct wim_ch_bw_param set frequency and bandwidth parameter for firmware

7 Power Save

IEEE802.11 SPEC defines power save to reduce power consumption on STA after Wi-Fi connection is done. A basic concept of power save is that STA enters PS (Power Save) mode if there is no TX and RX frames during a certain time and exits PS mode (in other word, enter AM (Active Mode)) when there is any buffered BU (Bufferable Unit) from AP or any frame to transmit. During power save, STA usually turns off power domains such as Baseband, RF, CPU, etc. (It is vendor specific). Linux mac80211 also implements basic operations of power save but it does not define which of power domains should be turned off during PS in detail because it is mainly Wi-Fi chipset dependent. The operations of Power Save that IEEE8021.11 defines are not described in the document, so please refer to IEEE802.11 SPEC, related books, or white papers.

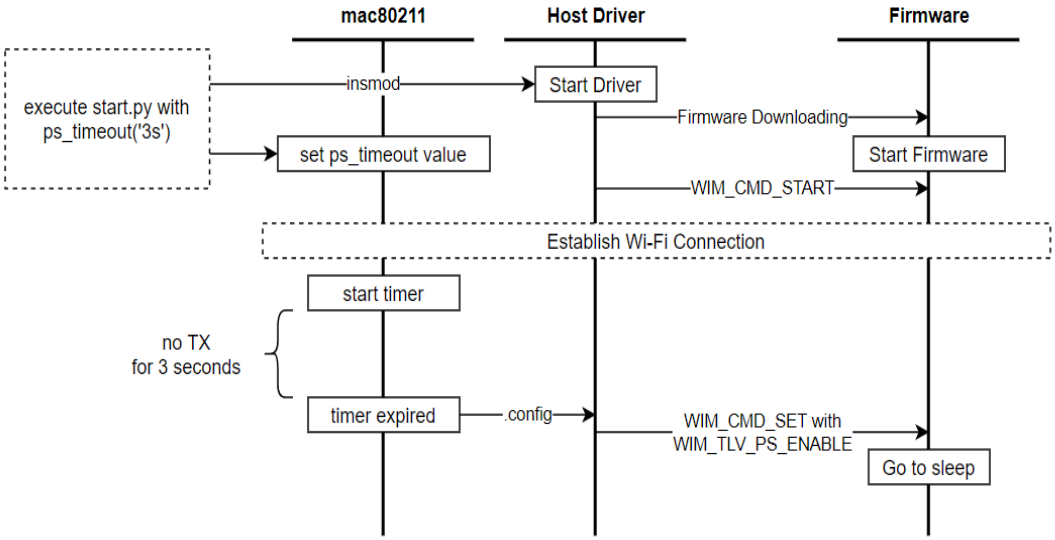

Figure 7.1 Entering power save

There are three or four major components related to PS operations, 1) mac80211 (Linux kernel module), 2) host NRC driver 3) target FW and/or 4) target uCode. Figure 7.1 ~ 7.6 show the operations and relationship among them while entering or exiting power save.

There are 2 parameters, 'ps_timeout' and 'sleep_duration', which are defined in 'start.py' script and very crucial for power save operation. First, 'ps_timeout' is timeout value in microsecond or second unit and STA enters power save if there is no traffic within this time. The value of this parameter is delivered to mac80211 through iw command and mac80211 starts its internal power save timer using this value after Wi-Fi connection is done (note. DHCP is not included in Wi-Fi Connection). Figure 7.1 shows this parameter is set while loading module. (i.e. insmod nrc.ko)
The other value 'sleep_duration' is also in microsecond or second unit and used as a module parameter. However, it is not for mac80211 but nrc host driver. The destination of this value is target FW where it is used as RTC timeout for chipset wakeup. (note. It is only for nonTIM deep sleep)

In summary, after Wi-Fi connection, mac80211 starts power save timer to check there is any frame to send or receive within 'ps_timeout'. If there is no frame within it, STA enters sleep. 'sleep_duration' is used to decide how long the target will fall asleep for nonTIM deep sleep.

7.1 Deep Sleep

Previously mentioned in 7.1, almost all the power domains (even including RAM) in target are off during Deep Sleep. However, target has small-size retention memory retained even in Deep Sleep and Wi-Fi connection information is stored in it before entering deep sleep. This information is used to quickly restore Wi-Fi connection after wake-up without exchanging any management frames with AP. It is also used as a space where a tiny program called uCode (micro code) runs to just check beacon and GPIO assert from host driver during Deep Sleep.

IEEE802.11ah SEPC defined a novel power save feature called non-TIM where STA does NOT check beacon periodically and just wakes up at certain time, and then sends PS-Poll frame or frame with PM 0 to check and receive BU from AP. Our target device also does not check beacon frames during non-TIM Deep Sleep and just wake-up according to external GPIO signal for TX or RTC timer.

There are three ways for STA to exit from Deep Sleep.

1. GPIO signal (for both TIM and nonTIM Deep Sleep)
   There is any traffic to send on host, NRC driver should make target wake. GPIO signal from host to target is used for this purpose.
2. RTC timer (only for nonTIM Deep Sleep)
   Target receives 'sleep_duration' value from NRC driver and falls asleep during 'sleep_duration'. Target device should wake up if timeout happens on it.
3. BU from AP (exactly bitmap of TIM element in Beacon) (only for TIM Deep Sleep)
   uCode in target device checks beacons periodically and wake-up if bitmap of its AID in beacon is set

As you have seen in Figure 7.1, target firmware is downloaded on target devices while loading NRC driver. This download procedure should be done after waking up because firmware on RAM was cleared during Deep Sleep. Figure 7.4 ~ 7.6 shows these procedures according to TIM and nonTIM Deep Sleep.

In TIM mode, NRC driver makes target wake by asserting GPIO on uCode to send frames. On the other hand, uCode triggers wake-up itself and then request for firmware download by asserting EIRQ to receive BU from AP. In nonTIM mode, target device can only wake up by RTC timer and GPIO signal for TX. It is because nonTIM STA does NOT check beacon during sleep.

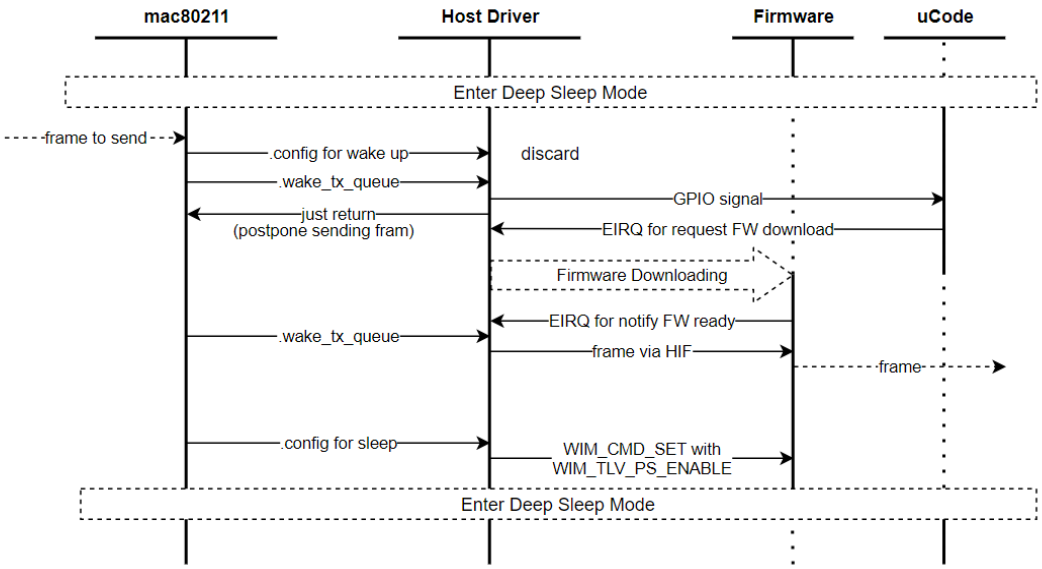

Figure 7.2 TX in Deep Sleep (common)

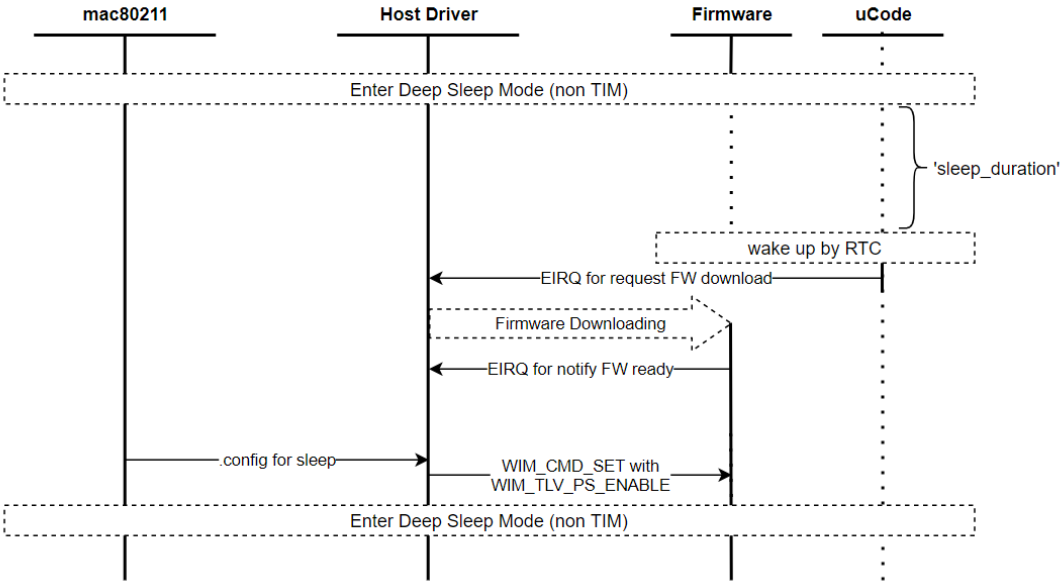

Figure 7.3 Wake up by RTC in Deep Sleep - non TIM

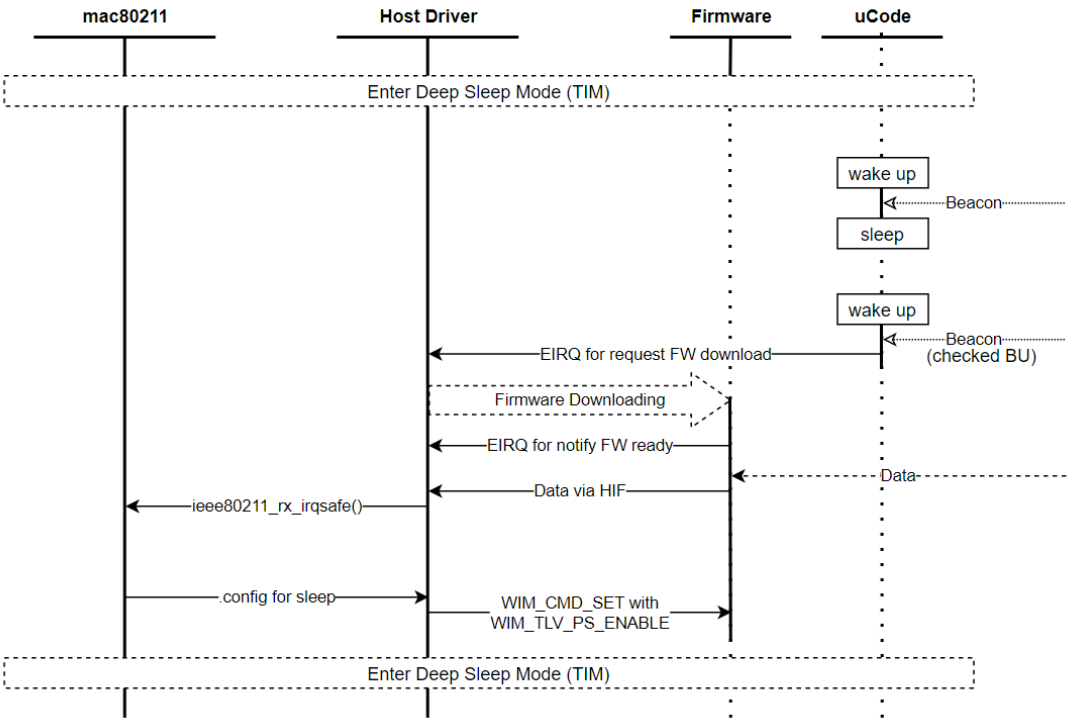

Figure 7.4 RX in Deep Sleep - TIM

7.2 Considering CQM procedure during sleep

According to CQM operation, mac80211 sends probe request every 30 seconds or it does not receive any beacons consecutively. (note. timeout and beacon count can be changed using module parameter of mac80211) If there is no probe response from AP, mac80211 disconnects current connection by sending de-authentication frame to AP and then tries to connect again by scanning AP. As mentioned in 7.1, CQM might be independent from power save, so host NRC driver build a probe response frame during sleep instead.

Detecting probe request frame on driver is done on *(*tx)* call-back function. Please refer to the driver codes for more detail implementation.

8 HSPI Register map

Table 8.1 Registers for HSPI

Address	Register	R/W	Description
System register			
0x00	WAKEUP	W	HSPI wakes up when writing 0x79
0x01	DEV_RESET	W	HSPI reset when writing 0xC8
IRQ register			
0x10	EIRQ_MODE	R/W	[07:03] Reserved [02] EIRQ_IO_EN (1: IO enable) [01] EIRQ_MODE1 (0: level trig., 1: edge trig.) [00] EIRQ_MODE2 (0: active low, 1: active high)
0x11	EIRQ_ENABLE	R/W	[07:04] Reserved [03] DEV_SLEEP_IRQ (1: enable) [02] DEV_READY_IRQ (1: enable) [01] TX_QUEUE_IRQ (1: enable) [00] RX_QUEUE_IRQ (1: enable)
0x12	EIRQ_CLEAR	R	EIRQ clear register Clear all interrupt when read
0x13	EIRQ_STATUS	R	[07:04] Reserved [03] DEV_SLEEP_STATUS [02] DEV_READY_STATUS [01] TX_QUEUE_STATUS [00] RX_QUEUE_STATUS
0x14	QUEUE_STATUS	R	[07:00] TX QUEUE status [47:40]
0x15			[07:00] TX QUEUE status [39:32]
0x16			[07:00] TX QUEUE status [31:24]
0x17			[07:00] TX QUEUE status [23:16]
0x18			[07:00] TX QUEUE status [15:08]
0x19			[07:00] TX QUEUE status [07:00]
0x1A			[07:00] RX QUEUE status [47:40]
0x1B			[07:00] RX QUEUE status [39:32]
0x1C			[07:00] RX QUEUE status [31:24]
0x1D			[07:00] RX QUEUE status [23:16]
0x1E			[07:00] RX QUEUE status [15:08]
0x1F			[07:00] RX QUEUE status [07:00]
Message register			
0x20	DEV_MSG_00	R	[07:00] Device message [31:24]
0x21			[07:00] Device message [23:16]

0x22			[07:00] Device message [15:08]
0x23			[07:00] Device message [07:00]
0x24	DEV_MSG_01	R	[07:00] Device message [31:24]
0x25			[07:00] Device message [23:16]
0x26			[07:00] Device message [15:08]
0x27			[07:00] Device message [07:00]
0x28	DEV_MSG_02	R	[07:00] Device message [31:24]
0x29			[07:00] Device message [23:16]
0x2A			[07:00] Device message [15:08]
0x2B			[07:00] Device message [07:00]
0x2C	DEV_MSG_03	R	[07:00] Device message [31:24]
0x2D			[07:00] Device message [23:16]
0x2E			[07:00] Device message [15:08]
0x2F			[07:00] Device message [07:00]
RX QUEUE register			
0x31	RXQUEUE_WINDOW	W	[07:00] received data from HSPI master (host)
TX QUEUE register			
0x41	TXQUEUE_WINDOW	R	[07:00] transmitted data to HSPI master (host)

9 Revision History

Revision No	Date	Comments
Ver 1.0	4/5/2023	Initial version

Appendix A.
Trouble shooting while insmod nrc_simple driver on Linux

A.1 Overview

This appendix provides how to handle the SPI-related problems when porting nrc driver to Linux host system. To handle it effeciently, we provide 'simple SPI-verification driver' by which the basic problems related to SPI can be found in easier way than using nrc driver (nrc.ko). You can find the simple driver including source codes in host/linux/driver/nrc_simple folder. (c.f. nrc driver is in host/linux/driver/nrc. Please do NOT confuse with them.)

We will introduce some error logs you might face while loading this driver via 'insmod `nrc_simple.ko`' and explain that what they mean and how to handle them.

The logs (by tail -f /var/log/syslog or dmesg) below can be seen if there is no SPI-related issue fortunately. (i.e. Both the register of spi driver and reading some registers are successfully done.)

```
[21515.066710] ### [nrc_simple] Value of paramters ###
[21515.066725] - bus_num: 0
[21515.066731] - chip_select: 0
[21515.066738] - max_speed_hz: 20000000
[21515.069661] [nrc_hspi_probe,L166]
[21515.069782] spi_sys_reg 00 01 72 92 00 00 00 01 01 02 07 16 de b0 97 57
[21515.070049] done successfully.
```

The simple driver supports three module parameters ('hifspeed', 'spi_bus_num' and 'spi_cs_num'). You can set appropriate values according to your host environment.

For example, '**sudo insmod nrc_simple.ko spi_bus_num=0 spi_cs_num=0 hifspeed=2000000**' means SPI device with maximum 20MHz Clock, 0 bus, 0 chip select is registered. You can remove module via '**sudo rmmod nrc_simple.ko**'

```
pi@raspberrypi:~ $ modinfo ./nrc_simple.ko
filename:       /home/pi/./nrc_simple.ko
description:    Newracom HSPI simple driver
license:        Dual BSD/GPL
author:         Newracom, Inc.(http://www.newracom.com)
srcversion:     A9961EFB30544D097E35E9E
depends:
name:           nrc_simple
vermagic:       4.19.97-v7 SMP mod_unload modversions ARMv7 p2v8
parm:           hifspeed:SPI master max speed (int)
parm:           spi_bus_num:SPI controller bus number (int)
```

parm:	**spi_cs_num:SPI chip select number (int)**

A.2 Case#1: could not find spi master with the bus number

- dmesg

```
[ 1991.867809] ### [nrc_simple] Value of paramters ###
[ 1991.867823] - bus_num: 1
[ 1991.867828] - chip_select: 0
[ 1991.867834] - max_speed_hz: 20000000
[ 1991.867845] [Error] could not find spi master with the bus number 1.
```

- Reason

  'bus number' represents a board-specific (and often SoC-specific) identifier for a given SPI controller. So, this number could be different according to how many controllers exist on the board or SoC. It can be verified by this command on Linux system.

  - $ls /sys/class/spi_master

```
debian@beaglebone:~$ ls /sys/class/spi_master
spi0  spi1
```

```
pi@raspberrypi:~ $ ls /sys/class/spi_master
spi0
```

  For example, 'Raspberry Pi B' supports only one SPI controller('spi0') but 'BeagleBone Black' supports two SPI controllers ('spi0' and 'spi1').

  So, the module parameter, 'spi_bus_num', should be assigned according to your host SPI controllers and then NRC7394 module or EVK can connect to SPI controller successfully.

A.3 Case#2: failed to instantiate a new spi device

- dmesg

```
[144.770625] ### [nrc_simple] Value of paramters ###
[144.770632] - bus_num: 0
[144.770638] - chip_select:
[144.770644] - max_speed_hz: 20000000
[144.770677] spi-bcm2835 3f204000.spi: chipselect 0 already in use
[144.770688] [Error] failed to instantiate a new spi device.
[153.791442] ### [nrc_simple] Value of paramters ###
[153.791458] - bus_num: 0
[153.791464] - chip_select: 1
[153.791473] - max_speed_hz: 20000000
[153.791531] spi-bcm2835 3f204000.spi: chipselect 1 already in use
[153.791548] [Error] failed to instantiate a new spi device.

[155.910897] ### [nrc_simple] Value of paramters ###
```

```
[155.910910] - bus_num: 0
[155.910915] - chip_select: 2
[155.910921] - max_speed_hz: 20000000
[155.910957] spi spi0.2: setup: only two native chip-selects are supported
[155.910971] spi-bcm2835 3f204000.spi: can't setup spi0.2, status -22
[155.910981] [Error] failed to instantiate a new spi device.
```

- Reason

  'chipselect # already in use' means that another SPI driver is already loaded for the SPI controller. In most cases, Linux system loads a default SPI driver after booting, which can access to SPI devices using normal userspace I/O calls. You can figure out which driver has already been loaded for the controller by the command below.

  - $ cat /sys/class/spi_master/spi0/spi0.0/modalias

```
pi@raspberrypi:~ $ cat /sys/class/spi_master/spi0/spi0.0/modalias
spi:spidev
pi@raspberrypi:~ $ cat /sys/class/spi_master/spi0/spi0.1/modalias
spi:spidev
```

  As you see, both 'chipselect 0' and 'chipselect 1' are used by 'spidev' driver as default on Raspberry Pi. You can also check whether that driver module has been loaded or not after boot.

  - $ lsmod | grep spidev

```
pi@raspberrypi:~ $ lsmod | grep spidev
spidev                 20480  0
```

In this case, you should disable or remove that 'spidev' driver to make nrc driver work correctly for the SPI master. There are a few methods for this.

1. Comment 'CONFIG_SPI_SPIDEV=m' from the kernel configuration file.
   (Please note that the kernel re-building is necessary.)

   - $ vi /lib/modules/`uname -r`/build/.config

```
#
# SPI Protocol Masters
#
CONFIG_SPI_SPIDEV=m
# CONFIG_SPI_LOOPBACK_TEST is not set
# CONFIG_SPI_TLE62X0 is not set
CONFIG_SPI_SLAVE=y
# CONFIG_SPI_SLAVE_TIME is not set
# CONFIG_SPI_SLAVE_SYSTEM_CONTROL is not set
# CONFIG_SPMI is not set
# CONFIG_HSI is not set
CONFIG_PPS=m
# CONFIG_PPS_DEBUG is not set
```

2. Rename 'spidev.ko'
   $ cd /lib/modules/`uname -r`/kernel/drivers/spi
   $ sudo mv spidev.ko spidev.ko.org

3. Add kernel module blacklist
   $ echo "blacklist spidev" >> /etc/modprobe.d/blacklist-spidev.conf
4. Device tree overlay
   (How to overlay device tree is out of scope but you can check the sample on RPi in document “UG-7394-018-Raspberry_Pi_setup”)

A.4 Case#3: failed to register spi driver

- dmesg

```
[268.288534] ### [nrc_simple] Value of paramters ###
[268.288548] - bus_num: 0
[268.288553] - chip_select: 0
[268.288559] - max_speed_hz: 20000000
[268.288871] done successfully.
[285.653480] ### [nrc_simple] Value of paramters ###
[285.653493] - bus_num: 0
[285.653498] - chip_select: 1
[285.653504] - max_speed_hz: 20000000
[285.653750] Error: Driver 'nrc-hspi-simple' is already registered, aborting...
[285.653757] [Error -16] failed to register spi driver(nrc-hspi-simple).
```

- Reason
  In this example, there is a driver which is already registered on chipselect 0. However, you can face such an error log like ‘-EBUSY(-16)’ if another driver with the same name tries to register for chipselect 1. To handle this issue, you have to change the driver’s name differently.

A.5 Case#4: invalid ACK after registering driver

- dmesg

```
[3981.573070] ### [nrc_simple] Value of paramters ###
[3981.573084] - bus_num: 0
[3981.573089] - chip_select: 0
[3981.573095] - max_speed_hz: 40000000
[3981.573396] [nrc_hspi_probe,L180]
[3981.573477] ------------[ cut here ]------------
[3981.573505] WARNING: CPU: 0 PID: 1642 at /home/pi/work/driver/nrc_hspi_simple_driver/nrc_hspi_simple_driver.c:165 c_spi_read_regs.constprop.0+0x170/0x190 [nrc_simple]
[3981.573511] Modules linked in: nrc_simple(O+) fuse 8021q garp stp llc binfmt_misc snd_bcm2835(C) raspberrypi_hwmon hwmon snd_pcm snd_timer bcm2835_codec(C) bcm2835_v4l2(C) v4l2_common v4l2_mem2mem snd videobuf2_vmalloc bcm2835_mmal_vchiq(C) videobuf2_dma_contig videobuf2_memops videobuf2_v4l2 i2c_bcm2835 videobuf2_common
```

```
spi_bcm2835 videodev media vc_sm_cma(C) uio_pdrv_genirq uio fixed mac80211 sha256_generic
cfg80211 rfkill i2c_dev ip_tables x_tables ipv6 [last unloaded: nrc_simple]
[3981.573628] CPU: 0 PID: 1642 Comm: insmod Tainted: G WC O 4.19.97-v7+ #1294
[3981.573633] Hardware name: BCM2835
[3981.573665] [<801120c0>] (unwind_backtrace) from [<8010d5f4>] (show_stack+0x20/0x24)
[3981.573683] [<8010d5f4>] (show_stack) from [<80845f28>] (dump_stack+0xe0/0x124)
[3981.573702] [<80845f28>] (dump_stack) from [<80120c9c>] (__warn+0x104/0x11c)
[3981.573718] [<80120c9c>] (__warn) from [<80120dec>] (warn_slowpath_null+0x50/0x58)
[3981.573740] [<80120dec>] (warn_slowpath_null) from [<7f5841a4>]
(c_spi_read_regs.constprop.0+0x170/0x190 [nrc_simple])
[3981.573792] [<7f5841a4>] (c_spi_read_regs.constprop.0 [nrc_simple]) from [<7f58422c>]
(nrc_hspi_probe+0x68/0xd8 [nrc_simple])
[3981.573816] [<7f58422c>] (nrc_hspi_probe [nrc_simple]) from [<8062404c>]
(spi_drv_probe+0x88/0xb4)
[3981.573836] [<8062404c>] (spi_drv_probe) from [<805d4cb8>] (really_probe+0x23c/0x2d4)
[3981.573852] [<805d4cb8>] (really_probe) from [<805d4f24>] (driver_probe_device+0x70/0x1ac)
[3981.573868] [<805d4f24>] (driver_probe_device) from [<805d5150>] (__driver_attach+0xf0/0xf4)
[3981.573883] [<805d5150>] (__driver_attach) from [<805d29e8>] (bus_for_each_dev+0x78/0xc4)
[3981.573898] [<805d29e8>] (bus_for_each_dev) from [<805d458c>] (driver_attach+0x2c/0x30)
[3981.573913] [<805d458c>] (driver_attach) from [<805d3f78>] (bus_add_driver+0x1ac/0x224)
[3981.573928] [<805d3f78>] (bus_add_driver) from [<805d593c>] (driver_register+0x8c/0x124)
[3981.573944] [<805d593c>] (driver_register) from [<80623f7c>] (__spi_register_driver+0x68/0x6c)
[3981.573965] [<80623f7c>] (__spi_register_driver) from [<7f5890d4>] (nrc_init+0xd4/0x1000
[nrc_simple])
[3981.573991] [<7f5890d4>] (nrc_init [nrc_simple]) from [<8010312c>]
(do_one_initcall+0x50/0x218)
[3981.574011] [<8010312c>] (do_one_initcall) from [<801ba19c>] (do_init_module+0x74/0x220)
[3981.574029] [<801ba19c>] (do_init_module) from [<801b9114>] (load_module+0x1dc0/0x2404)
[3981.574045] [<801b9114>] (load_module) from [<801b9974>] (sys_finit_module+0xbc/0xcc)
[3981.574060] [<801b9974>] (sys_finit_module) from [<80101000>] (ret_fast_syscall+0x0/0x28)
[3981.574067] Exception stack(0xa495dfa8 to 0xa495dff0)
[3981.574080] dfa0: ca4a5500 0002abf4 00000003 01c01150 00000000
00000013
[3981.574092] dfc0: ca4a5500 0002abf4 0003fce8 0000017b 01c02830 01c01150 00000003
01c01150
[3981.574101] dfe0: 7e96f5b8 7e96f5a8 00022cb8 76bf6af0
[3981.574109] ---[ end trace 8dec2453c4bbcc73 ]---
[3981.574115] [Error] failed to read register(0x0).
[3981.574141] nrc-hspi-simple: probe of spi0.0 failed with error -5
[3981.574224] done successfully.
```

- Reason
  This warning message is shown when SPI master cannot receive ACK or receive Invalid ACK

from SPI slave (on target side) after sending the read-register command. It might happen in case that

- 'hifspeed' is so high that NRC7394 HSPI HW cannot make signal synchronously. The maximum HSPI frequency of NRC7394 can be found in the datasheet.
- Wrong spi mode is set. Our target SPI only supports mode 'SPI_MODE_0 (CPOL 0, CPAH 0)' so you should set spi mode on master as 'SPI_MODE_0'

  (Please check your host data sheet for how to set SPI mode)
- Other HW issues. For example, wrong wiring between SPI master and slave. So, it would be necessary to verify the real input/output signal of the SPI pins using an equipment such as oscilloscope.

Please note that 'insmod' can be done successfully even though hifspeed is too high. It is because the very high 'hifspeed' does not affect driver-register work itself.

If nrc_simple driver is loaded successfully, basic SPI register and operation are verified. So next stage is to verify SPI performance via loopback test (refer to Appendix B in this document) and Wi-Fi throughput using start STA and AP with start.py script (refer to UG-7394-001-EVK User Guide (Host Mode))