NL Radio Driver API Document

Doc. No. 001-70689 Rev. *B

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
http://www.cypress.com

Copyrights

Trademarks

Source Code

Disclaimer

Contents

1. Introduction

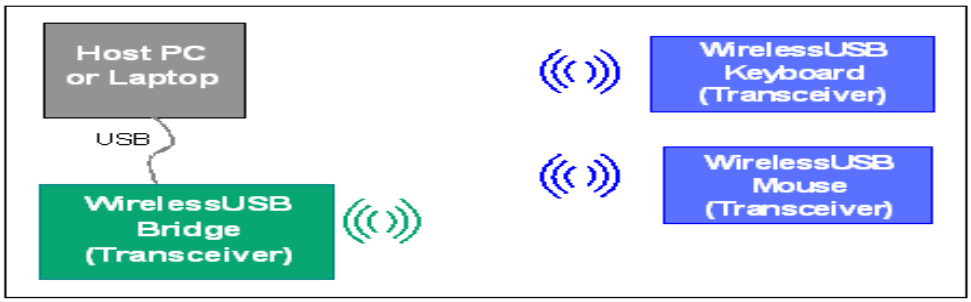
The NL radio driver provides users with a consistent interface to the NL radio. The driver is designed to interface with both C and assembly written applications and consists of the following files:

- Nlradio.asm
- Nlradio.h
- Nlradio.inc
- Nlspi.asm

This document describes the APIs exposed by WUSB-NL driver.

The NL radio driver is used in wireless mouse, keyboard, and bridge application software stacks. The NL radio driver is modular and can be used as a library. The API exported by this module is explained in this document. The following is a system level block diagram of a typical wireless mouse and keyboard application.

Figure 1-1. System Level Block Diagram



References

1. 001-13683 Rev. *B - SPI Master Data Sheet SPIM V 1.20
2. 001-13678 Rev. *F – SPI Master Data Sheet SPIM V 2.5

Document Revision History

Revision	Issue Date	Origin of Change	Description of Change
**	07/22/2011	DATT	New spec.
*A	08/19/2011	KPMD	Changed posting to external web.
*B	12/13/2011	KNTH	Update to match driver v1.4.

2. Description

API Use

This section describes high level use of APIs. It also describes the requirement to use this module and the type declaration used within this module.

2.1.1 Initialization

Before the radio can be used, it must be initialized using the API call RadioInit.

Example usage:

```
RadioInit();
```

2.1.2 Transmitting

There are two types of transmit function: blocking and non blocking. Blocking transmit does not return until the transmit process has completed. The non blocking function starts transmit and then returns. It is the responsibility of the calling application to monitor the start of transmits and terminate when necessary. Both forms of transmits require a call to RadioSetPtr with a buffer address as a parameter. This pointer points to the start of the buffer to be transmitted.

Transmit examples:

Blocking	**Non Blocking**
RadioSetPtr	RadioSetPtr
RadioBlockingTransmit	RadioStartTransmit
	RadioGetTransmitState
	RadioEndTransmit

2.1.3 Receiving

By default, receives are non blocking and require a similar set of calls as used in the non blocking transmit functions. RadioAbort must be called to abort a receive operation. Note that after receive starts, no calls can be made to the configuration access routines until the receive operation is terminated.

RX examples:

- RadioSetPtr
- RadioStartReceive
- RadioGetReceiveState
- RadioEndReceive

Requirements

2.1.4 Header files

To use the radio driver you must include *nlRadio.h* or *nlRadio.inc* in any file that calls radio driver functions.

2.1.5 HW interface

The SPI Master block of the PSoC Designer UM (refer [1]) used to interface to the radio is named: SPIM_Radio. This block provides the firmware interface to the SPI pins, so their names have no requirements.

2.1.6 Pins

In the PSoC Designer workspace, name the pins as mentioned in the following table.

Sl. No	Pin Description	Pin Name
1	Pin connected to NL Radio's Slave select pin	NL_nSS
2	Pin connected to NL Radio's reset pin	NL_RST
3	Pin connected to BIND button	BIND_BUTTON
4	SPI Pins to be prefixed with SPIM_Radio_	

Type Declarations and Definitions

The type declarations are as follows. These declarations are kept the same as the LP radio driver to keep the changes in existing protocol and application to minimum.

BYTE: Used for 8-bit register values.

WORD: Used for 16-bit register values.

RADIO_CONST_PTR: Used for ROM buffer pointers.

RADIO_BUFFER_PTR: Used for RAM buffer pointers.

RADIO_LENGTH: Used for radio field lengths.

RADIO_REG_ADDR: 8 bit value used for Radio registers address.

RADIO_STATE: Type unsigned char to store the radio state. Radio States are

RADIO_IDLE 0x00

RADIO_RX 0x80

RADIO_TX 0x20

RADIO_DATA 0x02

RADIO_COMPLETE 0x04

RADIO_ERROR 0x08

RADIO_SLEEP 0x40

RADIO_RX_STATUS: Type unsigned char. Radio receive status are

RADIO_BAD_CRC 0x08

RADIO_BUF_NOT_SUF 0x30

RADIO_SUCCESS 0x80

	RADIO_FAILURE	0x00

XACT_CONFIG: Type unsigned int

ACK_EN	0x0800

ACK_TO is of following value

ACK_TO_4X	0x6B
ACK_TO_8X	0x9C
ACK_TO_12X	0xCD
ACK_TO_15X	0xFF

TX_CONFIG: Type unsigned int

PA_VAL is of following value

PA_N12_DBM	0x1E20
PA_N8_DBM	0x1C20
PA_N3_DBM	0x1A20
PA_0_DBM	0x1920
PA_1_DBM	0x1820

RADIO_FRAME_CONFIG : Type unsigned char

SOP_EN	0x00 This is ignored for NL
LEN_EN	0x20

RADIO_RSSI: Type unsigned char

RADIO_LENGTH: Type unsigned char

3. Radio High Level Functions

RadioInit

```
void RadioInit(void);
```

Parameters:

None

Return Value:

None

Description:

This function initializes the radio module. Before calling this function, initialize SPI Master with the following configurations:

- enCoR2 controller– BitOrder – MSB First, CPOL – Low, CPHA – High
- enCoRe5 controller – SPIM_Radio_SPIM_MODE_1, SPIM_Radio_SPIM_MSB_FIRST

For the definition of SPIM configuration refer to [1] and [2].

RadioSetChannel

```
void RadioSetChannel(BYTE channel);
```

Parameters:

channel 8-bit value that is passed to the function.

Return Value:

None

Description:

This function sets the radio channel to a specified frequency. The frequency has the value of (2402 + channel) MHz . This function takes an 8-bit argument that is passed to it by ‘BYTE channel’. RadioSetChannel is limited to a maximum of RADIO_MAX_CHAN, which is 78.

Example:

```
RadioSetChannel(10); // Put carrier at 2412MHz
```

RadioGetChannel

```
BYTE RadioGetChannel(void);
```

Parameters:

None

Return Value:

This function returns the 8-bit value of the channel number. The frequency should be interpreted as (2402 + channel) MHz.

Description:

This function gets an 8-bit value from the channel.

Example:

```
RadioSetFrequency(10); // Put carrier at 2412MHz
c = RadioGetChannel(); // Returns 10.
```

RadioSetTxConfig

```
void RadioSetTxConfig(TX_CONFIG config);
```

Parameters:

config: Radio power amplifier gain setting

Possible values for ‘config’ are:

PA_N19_DBM : -19dBm

PA_N12_DBM: -12dBm

PA_N8_DBM: -8dBm

PA_N3_DBM: -3dBm

PA_0_DBM: 0dBm

PA_1_DBM: 1dBm

Return Value:

None

Description:

This function sets the radio power amplifier gain.

Example:

```
RadioSetTxConfig(PA_N15_DBM); // Set PA gain to -15dBm
```

RadioGetTxConfig

```
TX_CONFIG RadioGetTxConfig(void);
```

Parameters:

None

Return Value:

Radio power amplifier gain setting. Possible values are:

PA_N19_DBM : -19dBm

PA_N12_DBM: -12dBm

PA_N8_DBM: -8dBm

PA_N3_DBM: -3dBm

PA_0_DBM: 0dBm

PA_1_DBM: 1dBm

Description:

This function returns the radio power amplifier gain.

Example:

```
RadioSetTxConfig(PA_N15_DBM);
if (RadioGetTxConfig() == PA_N3_DBM)
{
   /* Not entered */
}
```

RadioSetXactConfig

```
void RadioSetXactConfig(XACT_CONFIG config);
```

Parameters:

config: transaction configuration composed of two parts

a) Enable auto-ack

Possible value is ACK_EN

b) Rx ack timeout

Possible values are,

ACK_TO_4X

ACK_TO_8X

ACK_TO_12X

ACK_TO_15X

Return Value:

None

Description:

This function sets the transaction configuration composed of two parts, auto-ack enable and Rx ack timeout.

Example:

```
// Enable auto-ack and set maximum Rx ack timeout
RadioSetXactConfig(ACK_EN | ACK_TO_15X);
// Disable auto-ack and set maximum Rx ack timeout
RadioSetXactConfig(ACK_TO_15X);
```

RadioGetXactConfig

```
XACT_CONFIG RadioGetXactConfig(void);
```

Parameters:

None

Return Value:

Transaction configuration composed of two parts

a) Enable auto-ack

Possible value is ACK_EN

b) Rx ack timeout

Possible values are,

ACK_TO_4X

ACK_TO_8X

ACK_TO_12X

ACK_TO_15X

Description:

This function returns the transaction configuration composed of two parts, auto-ack enable and Rx ack timeout.

Example:

```
// Turn off the auto-ack function but keep the current ack timeout setting
RadioSetXactConfig(RadioGetXactConfig() & ~ACK_EN);
```

RadioSetFrameConfig

```
void RadioSetFrameConfig(RADIO_FRAME_CONFIG config);
```

Parameters:

config: frame configuration composed of

a) Syncword length.

Possible values are,

SYNC_WRD_64_BITS

SYNC_WRD_48_BITS

SYNC_WRD_32_BITS

SYNC_WRD_16_BITS

b) Packet length enable

Possible value is LEN_EN

Return Value:

None

Description:

This function sets the frame configuration composed of two parts, packet length enable and syncword length. Syncword length can be 64, 48, 32 or 16 bits. Enabling packet length causes the radio to treat the first byte of the payload as the length.

Example:

```
// Set syncword length to 48 bits and enable packet length.
RadioSetFrameConfig(SYNC_WRD_48_BITS | LEN_EN);
```

RadioGetFrameConfig

```
RADIO_FRAME_CONFIG RadioGetFrameConfig(void);
```

Parameters:

None

Return Value:

Frame configuration composed of

a) Syncword length.

Possible values are,

SYNC_WRD_64_BITS

SYNC_WRD_48_BITS

SYNC_WRD_32_BITS

SYNC_WRD_16_BITS

b) Packet length enable

Possible value is LEN_EN

Description:

This function sets the frame configuration composed of two parts, packet length enable and syncword length. Syncword length can be 64, 48, 32 or 16 bits. Enabling packet length causes the radio to treat the first byte of the payload as the length.

Example:

```
// Turn on packet length but keep the current syncword setting
RadioSetFrameConfig(RadioGetFrameConfig() | LEN_EN);
```

RadioSetPreambleCount

```
void RadioSetPreambleCount(BYTE count);
```

Parameters:

count: preamble length

Possible values are,

PREAMBLE_LEN_1_BYTE

PREAMBLE_LEN_2_BYTE

PREAMBLE_LEN_3_BYTE

PREAMBLE_LEN_4_BYTE

PREAMBLE_LEN_5_BYTE

PREAMBLE_LEN_6_BYTE

PREAMBLE_LEN_7_BYTE

PREAMBLE_LEN_8_BYTE

Return Value:

None

Description:

This function sets the preamble length in bytes. The maximum length of the preamble is 8.

Example,

```
// Set the preamble length to 1 byte
RadioSetPreambleCount(PREAMBLE_LEN_1_BYTE);
```

RadioGetPreambleCount

```
BYTE RadioGetPreambleCount(void);
```

Parameters:

None

Return Value:

The preamble length

Description:

This function gets the preamble length in bytes. The maximum length of the preamble is 8.

RadioSetCrcSeed

```
void RadioSetCrcSeed(WORD crcSeed);
```

Parameters:

crcSeed The crcSeed value.

Return Value:

None

Description:

This function sets the value used as the CRC seed value for both transmit and receive. As the NL supports only 8-bit CRC, the MSB of crcSeed is ignored.

RadioGetCrcSeed

```
WORD RadioGetCrcSeed(void);
```

Parameters:

None

Return Value:

CRC seed value for both transmit and receive. As the NL supports only 8-bit CRC, the MSB of the returned CRC seed is always 0.

Description:

Returns the current CRC seed value. As the NL supports only 8-bit CRC, the MSB of CRC seed is ignored.

RadioSetPtr

```
void RadioSetPtr(RADIO_BUFFER_PTR ramPtr);
```

Parameters:

RamPtr: Pointer to RAM buffer for future operations.

Return Value:

None

Description:

Set the buffer pointer address for future transmit and receive operations.

RadioSetLength

```
void RadioSetPtr(RADIO_LENGTH length);
```

Parameters:

length: Length of buffer pointed to by most recent call to RadioSetPtr.

Return Value:

None

Description:

Set the buffer length pointed to by most recent call to RadioSetPtr.

Example,

```
RadioSetLength(sizeof(rx_packet));
RadioSetPtr((unsigned char *)&rx_packet);
RadioStartReceive();
```

RadioStartTransmit

```
void RadioStartTransmit(BYTE retryCount, RADIO_LENGTH length);
```

Parameters:

retryCount: Number of retries.

length: Number of bytes to transmit.

Return Value:

None

Description:

Start the non blocking transmission of a packet. The location of the packet buffer to transmit must have previously been set with a call to RadioSetPtr.

After starting the transmission of a packet with this call, the state of the transmit operation should be checked by calling RadioGetTransmitState. When RadioGetTransmitState indicates that the transmission has completed, a call should be made to RadioEndTransmit.

After calling RadioStartTransmit, no calls can be made to the configuration access routines until the transmit operation is terminated with a call to RadioEndTransmit or RadioAbort. Until one of those calls is made to end the transmit operation, the only other call supported is RadioGetTransmitState.

Example:

```
unsigned char bufferToTransmit[14] = “PacketPayload”;
RadioSetPtr(bufferToTransmit);
RadioStartTransmit(0, sizeof(bufferToTransmit));
while (RadioGetTransmitState() != RADIO_COMPLETE)
{
        CallSomeRoutineToGetWorkDoneWhileWaitingForTransmitToComplete();
}
RadioEndTransmit();
```

RadioGetTransmitState

```
RADIO_STATE RadioGetTransmitState(void);
```

Parameters:

None

Return Value:

Returns the state of the current transmit operation.

Description:

This call should be made after starting a transmit operation with the RadioStartTransmit function. The value returned is of the type RADIO_STATE and can take a value of RADIO_COMPLETE or RADIO_TX.

RadioEndTransmit

```
void RadioEndTransmit(void);
```

Parameters:

None

Return Value:

None

Description:

Completes a transmit operation.

RadioBlockingTransmit

```
RADIO_STATE RadioBlockingTransmit(BYTE retryCount, RADIO_LENGTH length);
```

Parameters:

retryCount: Number of times the packet should be retried it the transmit fails.

length: Length, in bytes, of the pocket.

Return Value:

```
Radio state.
```

Description:

Transmit a packet. Block execution until it completes. This function attempts to transmit a packet. The address of the packet buffer should have previously been set with a call to RadioSetPtr.

Example:

```
unsigned char bufferToTransmit[14] = “PacketPayload”;
RADIO_STATE txState;
RadioSetPtr(bufferToTransmit);
txState = RadioBlockingTransmit(4, sizeof(bufferToTransmit));
if (txState == RADIO_ERROR)
        printf(“Transmit failed after 5 attempts.\r\n”);
```

RadioStartReceive

```
void RadioStartReceive(void);
```

Parameters:

None

Return Value:

None

Description:

Start the reception of a packet. The location and length of the packet buffer to receive data must have previously been set with calls to RadioSetPtr and RadioSetLength.

After starting the reception of a packet with this call, the state of the receive operation should be checked by calling RadioGetReceiveState. When RadioGetReceiveState indicates that the transmission has completed, a call should be made to RadioEndReceive.

After calling RadioStartReceive, no calls can be made to the configuration access routines until the receive operation is terminated with a call to RadioEndReceive or RadioAbort.

Until one of those calls is made to end the receive operation, the only other calls supported are RadioGetReceiveState and RadioGetRssi.

RadioGetReceiveState

```
RADIO_STATE RadioGetReceiveState(void);
```

Parameters:

None

Return Value:

Returns the state of the current receive operation.

Description:

This call should be made after starting a receive operation with the RadioStartReceive function.

This function checks for errors during reception or completion of reception and returns either RADIO_ERROR or RADIO_COMPLETE. If reception was completed with no errors, received data would be available in the buffer pointed to by the most recent call to RadioSetPtr.

Example**:**

```
RadioSetLength(sizeof(rx_pkt));
RadioSetPtr((unsigned char *)&rx_pkt);
RadioStartReceive();
while (1)
{
        M8C_ClearWDTAndSleep;
        status = RadioGetReceiveState();
        if (status == RADIO_COMPLETE)
        {
                pkt_len = RadioEndReceive();
                ProcessPacket();
                RadioSetLength(sizeof(rx_pkt));
                RadioSetPtr((unsigned char *)&rx_pkt);
                RadioStartReceive();
        }
        else if (status == RADIO_ERROR)
        {
```

```
                        (void)RadioEndReceive();
                         HandleErrors();
                         RadioSetLength(sizeof(rx_pkt));
                         RadioSetPtr((unsigned char *)&rx_pkt);
                    RadioStartReceive();
               }
          }
```

RadioEndReceive

```
RADIO_LENGTH RadioEndReceive(void);
```

Parameters:

None

Return Value:

Returns the length of the packet that was received. If packet payload truncation occurs (due to inadequate buffer length), it does not change this return value.

Description:

Completes a receive operation.

RadioForceState

```
void RadioForceState(XACT_CONFIG endStateBitsOnly);
```

Parameters:

endStateBitsOnly: Immediate new state for the radio.

Return Value:

None

Description:

This function immediately changes the radio state. If the Radio is in Transmit or Receive modes, then RadioAbort MUST be called before calling RadioForceState.

Example:

```
RadioForceState(END_STATE_SLEEP);
```

RadioGetRssi

```
RADIO_RSSI RadioGetRssi(void);
```

Parameters:

None

Return Value:

RADIO_RSSI – Received packet signal strength or carrier strength.

Description:

This function returns the Receive Signal Strength Indicator value. When read completes, the return value can be the signal strength of the received packet or of the noise. If the MSB of the return value is 0, then the signal strength is for the received packet; if not, it is for noise. While in the receive state, but before a packet has been received, this can be called continuously to check the carrier strength at the current frequency. It is necessary that RadioGetRssi be called at least 100us after the radio entered Receive mode.

RadioAbort

```
RADIO_LENGTH RadioAbort(void);
```

Parameters:

None

Return Value:

Zero.

Description:

Aborts a transmit or receive operation and transitions the radio to the idle state. This function always returns a zero.

RadioState

```
RADIO_STATE RadioState;
```

Parameters:

None

Return Value:

None

Description:

RadioState is a global variable that contains the state of the radio and the driver.

The values for RadioState:

RADIO_IDLE: The radio is neither in Receive mode nor in Transmit mode.

RADIO_RX: The radio is in Receive mode.

RADIO_TX: The radio is in Transmit mode.

RADIO_COMPLETE: The radio has successfully completed a transmit or receive operation.

RADIO_ERROR: The radio transmit or receive operation was unsuccessful.