WUSB-NL Radio Driver API Guide

Doc. No. 001-70689 Rev. *C

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

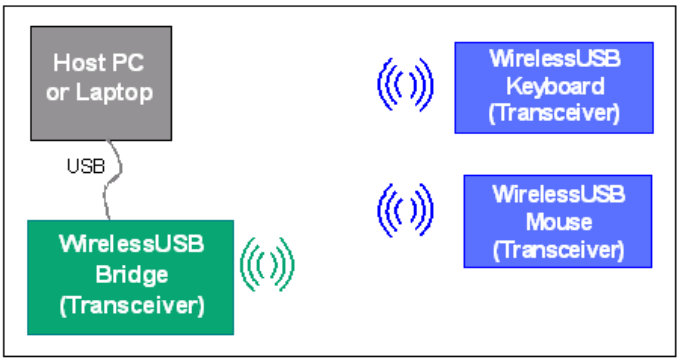
The WUSB-NL radio driver provides users with a consistent interface to the WUSB-NL radio. The driver is designed to interface with both C and M8C assembly written applications and consists of the following files:

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

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

The WUSB-NL radio driver is used in wireless mouse, keyboard, and bridge application software stacks. The WUSB-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



1.1 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.
*C	11/28/2012	PRAM	Update to match driver v2.0 Updated document title.

2 Description

2.1 API Use

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

2.1.1 Initialization

RadioInit initializes the radio and should be called before any other function. Before calling this function the SPI Master connected to WUSB-NL radio must be initialized according to the configuration specified in the radio datasheet. Delay functions (`TimerDelay10usec and TimerDelay100usec`) are required by driver and should be initialized before initializing radio.

Initialization example:

```
RadioInit();
```

2.1.2 Transmitting

RadioBlockingTransmit is the transmit function provided by the driver. A call to RadioSetPtr and RadioSetLength is required before calling RadioBlockingTransmit This function loads the data into the FIFO and monitors the status of the radio. It will return RADIO_COMPLETE if the radio transmission is successful and RADIO_ERROR otherwise.

Transmit example:

```
unsigned char state,txbuf[4] = {0x5a,0x5a,0x5a,0x5a};
RadioSetPtr(txbuf);
RadioSetLength(sizeof(txbuf));
state = RadioBlockingTransmit(1, sizeof(txbuf));
if (bState & RADIO_COMPLETE) { //Take some action; };
else if (bState & RADIO_ERROR) { //Take some action; };
```

2.1.3 Receiving

RadioStartReceive should be called in order to start a receive operation. A call to RadioSetPtr and RadioSetLength are required before calling RadioStartReceive. After calling RadioStartReceive, RadioGetReceiveState should be polled to get the status of the receive operation. RadioGetReceiveState will return RADIO_RX, RADIO_COMPLETE or RADIO_ERROR. A return value of RADIO_RX indicates that the receive operation is in progress. A return value of RADIO_COMPLETE indicates that the receive operation is complete and data is available in the buffer specified by the call to RadioSetPointer. A return value of RADIO_ERROR indicates that there was an error during the receive operation. RadioAbort or RadioEndReceive must be called to stop 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:

```
BYTE arbRxData[7];
```

```
RadioSetLength(sizeof(arbRxData));
RadioSetPtr((unsigned char *)&arbRxData);
RadioStartReceive();
do {
      // Other system tasks
      bState = RadioGetReceiveState();
} while((bState & (RADIO_ERROR | RADIO_COMPLETE)) == 0);
RadioEndReceive();
if (bState & RADIO_COMPLETE) { //Take some action; };
if (bState & RADIO_ERROR) { //Action; };
```

2.2 Requirements

2.2.1 Header files

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

2.2.2 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.

2.2.3 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 WUSB-NL Radio'sSlave select pin	NL_nSS
2	Pin connected to WUSB-NL Radio'sreset pin	NL_RST
4	SPI Pins to be prefixed with SPIM_Radio_	SPIM_Radio_MOSI SPIM_Radio_SCLK SPIM_Radio_MISO

2.3 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
	END_STATE_SLEEP	0x00
	END_STATE_IDLE	0x01

RADIO_RX_STATUS: Type unsigned char. Radio receive status are

RADIO_BAD_CRC	0x08
RADIO_BUF_NOT_SUF	0x30
RADIO_GFSK	0x00

FUN_STATUS: Type unsigned char

RADIO_SUCCESS	0x80
RADIO_FAILURE	0x00
RADIO_ABORT_SUCCESS	0xFF

XACT_CONFIG: Type unsigned int

ACK_EN	0x0800
ACK_TO_MSK	0xFF

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_N19_DBM	0x3FA0
PA_N12_DBM	0x1E20
PA_N8_DBM	0x1C20
PA_N3_DBM	0x1A20
PA_0_DBM	0x1920
PA_1_DBM	0x1820

PA_VAL can be masked by

PA_VAL_MSK	0xFFFF

RADIO_FRAME_CONFIG: Type unsigned char

SOP_EN	0x00 // This is ignored for NL
LEN_EN	0x20
SYNC_WRD_MSK	0x18
SYNC_WRD_64_BITS	0x18
SYNC_WRD_48_BITS	0x10
SYNC_WRD_32_BITS	0x08
SYNC_WRD_16_BITS	0x00
SOP_THRESH_MSK	0x1F

RADIO_RSSI: Type unsigned char

NOISE_RSSI	0x80
RSSI_LVL_MSK	0x3F

RADIO_LENGTH: Type unsigned char

Preamble definitions are as follows

PREAMBLE: Type unsigned char

PREAMBLE_LEN_MSK	0xE0
PREAMBLE_LEN_1_BYTE	0x00
PREAMBLE_LEN_2_BYTE	0x20
PREAMBLE_LEN_3_BYTE	0x40
PREAMBLE_LEN_4_BYTE	0x60
PREAMBLE_LEN_5_BYTE	0x80
PREAMBLE_LEN_6_BYTE	0xA0
PREAMBLE_LEN_7_BYTE	0xC0
PREAMBLE_LEN_8_BYTE	0xE0

NL Register definitions are as follows

ANALOG_BLK_CONFIG1_ADR	0
ANALOG_BLK_CONFIG2_ADR	1
RX_CTRL_ADR	2
TX_RX_STS_ADR	3
BPF_CTRL1_ADR	4
BPF_CTRL2_ADR	5
RSSI_VAL_ADR	6

```
RF_SYNTH_TX_RX_CTRL_ADR       7
TX_DAC_CTRL_ADR               8
PA_CTRL_ADR                   9
AMS_TST_CTRL1_ADR             10
AMS_TST_CTRL2_ADR             11
AMS_TST_CTRL3_ADR             12
RX_IF_CTRL_ADR                13
PM_CTRL_ADR                   14
RSRVD_CTRL1_ADR               15
RSRVD_CTRL2_ADR               16
RSRVD_CTRL3_ADR               17
RSRVD_CTRL4_ADR               18
RSRVD_CTRL5_ADR               19
RSRVD_CTRL6_ADR               20
RSRVD_CTRL7_ADR               21
PWRMGMT_ADR                   22
TX_RX_VCO_CAL_CTRL_ADR        23
FRACT_N_VCO_CAL_CTRL1_ADR     24
FRACT_N_VCO_CAL_CTRL2_ADR     25
VCO_SET_CTRL_ADR              26
VCO_N_CTYST_TRIM_ADR          27
FREQ_VAL_ADR                  28
MAN_REV_CODE_ADR              29
MAN_ID_LSB_ADR                30
MAN_ID_MSB_ADR                31
CONFIG_ADR                    32
DELAY_REG0_ADR                33
DELAY_REG1_ADR                34
MISC1_ADR                     35
SYNC_WORD1_ADR                36
SYNC_WORD2_ADR                37
SYNC_WORD3_ADR                38
SYNC_WORD4_ADR                39
THRESHOLD_REG_ADR             40
MISC2_ADR                     41
MISC3_ADR                     42
```

RSRVD_CTRL8_ADR	43
RSRVD_CTRL9_ADR	44
RSRVD_CTRL10_ADR	45
RSRVD_CTRL11_ADR	46
RSRVD_CTRL12_ADR	47
MAIN_STS_REG_ADR	48
RSRVD_CTRL13_ADR	49
TX_RX_FIFO_REG_ADR	50
RSRVD_CTRL14_ADR	51
FIFO_RD_PTR_REG_ADR	52

3 Radio High Level Functions

3.1 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:

- enCoRe2 controller– BitOrder – MSB First, CPOL – Low, CPHA – High
- enCoRe5 controller – SPIM_Radio_SPIM_MODE_2, SPIM_Radio_SPIM_MSB_FIRST

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

3.2 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 value of RADIO_MAX_CHAN, which is defined to 78.

Example:

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

3.3 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:

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

3.4 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_N3_DBM); // Set PA gain to -3dBm
```

3.5 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_0_DBM);
if (RadioGetTxConfig() != PA_0_DBM)
{
   /* Not entered */
}
```

3.6 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      // Wait for 107 us
ACK_TO_8X      // Wait for 156 us
ACK_TO_12X     // Wait for 205 us
ACK_TO_15X     // Wait for 255 us
```

Return Value:

None

Description:

This function sets the transaction configuration composed of two parts, auto-ack enable and Rx ack timeout. Auto-ack is used to send ACK after receiving the data.

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);
```

3.7 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);
```

3.8 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. After setting Syncword length actual sync words are to be set by upper layer. 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);
```

3.9 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);
```

```
if (RadioGetFrameConfig() == SYNC_WRD_48_BITS)
 {
     /* Enters if above condition is satisfied */
 }
```

3.10 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);
```

3.11 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.

Example:

```
if (RadioGetPreambleCount() == PREAMBLE_LEN_1_BYTE)
{
    /* Enters if the above condition is satisfied */
}
```

3.12 RadioSetCrcSeed

```
void RadioSetCrcSeed(BYTE crcSeed);
```

Parameters:

`crcSeed` The crcSeed value (0 to 255).

Return Value:

None

Description:

This function sets the value used as the CRC seed value for both transmit and receive. The WUSB-NL radio supports a 16-bit CRC with an 8 bit CRC seed.

3.13 RadioGetCrcSeed

```
BYTE RadioGetCrcSeed(void);
```

Parameters:

None

Return Value:

CRC seed value for both transmit and receive.

Description:

Returns the current CRC seed value. The WUSB-NL radio supports a 16-bit CRC with an 8 bit CRC seed.

3.14 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.

Example:

`RADIO_BUFFER_PTR` arbTxData = {1,2,3,4,5,6,7}; // Initialize the transmission buffer

RadioSetPtr(arbTxData);

3.15 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();
```

3.16 RadioBlockingTransmit

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

Parameters:

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

length: Length, in bytes, of the packet.

Return Value:

```
Radio state.
```

Description:

This function transmits a packet. Blocks execution until it completes. 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_COMPLETE)
       printf("Transmit successful.\r\n");
if (txState == RADIO_ERROR)
        printf("Transmit failed after 5 attempts.\r\n");
```

3.17 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.

3.18 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 RADIO_ERROR, RADIO_COMPLETE or RADIO_RX. 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)
 {
        // Other system tasks
        status = RadioGetReceiveState();
        if (status == RADIO_COMPLETE)
        {
                pkt_len = RadioEndReceive();
```

```
                    RadioSetLength(sizeof(rx_pkt));
                    RadioSetPtr((unsigned char *)&rx_pkt);
                    RadioStartReceive();
                }
                else if (status == RADIO_ERROR)
                {
                    (void)RadioEndReceive();
                     RadioSetLength(sizeof(rx_pkt));
                     RadioSetPtr((unsigned char *)&rx_pkt);
                   RadioStartReceive();
                }
            }
```

3.19 RadioEndReceive

RADIO_LENGTH RadioEndReceive(void);

Parameters:

None

Return Value:

This function returns the length of the received packet. In the case of packet payload truncation (due to inadequate buffer length), the return value is zero.

Description:

Completes a receive operation.

3.20 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 receive mode, then RadioAbort or RadioEndReceive must be called before calling RadioForceState.

Example:

```
RadioForceState(END_STATE_SLEEP);
```

3.21 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. It is necessary that RadioGetRssi be called at least 100us after the radio has entered Receive mode.

3.22 RadioAbort

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

Parameters:

None

Return Value:

Zero.

Description:

Aborts a transmit or receive operation and transitions the radio to the idle state.

3.23 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.

`RADIO_SLEEP:` The radio is in sleep mode.