UZ2400

Silicon Version D

Low Power 2.4 GHz Transceiver for IEEE 802.15.4 Standard

Sample Code (A Supplement to UZ2400 Datasheet Section 4)

AN-2400-78

The content of this technical document is subject to change without notice. Please contact UBEC for further information.

Version: 0.0
Released Date: 2009/11/30

UZ2400

Low Power 2.4 GHz Transceiver for IEEE 802.15.4 Standard

1. Introduction

This sample code is a simple function code base on the function descriptions given in Section 4 of UZ2400 datasheet. It is MCU-independent and can be ported to various hardware platforms according to the designer's preference.

Note: The sample code will use an SPI driver which is hardware dependent for different MCUs.

2. Data Type Definitions

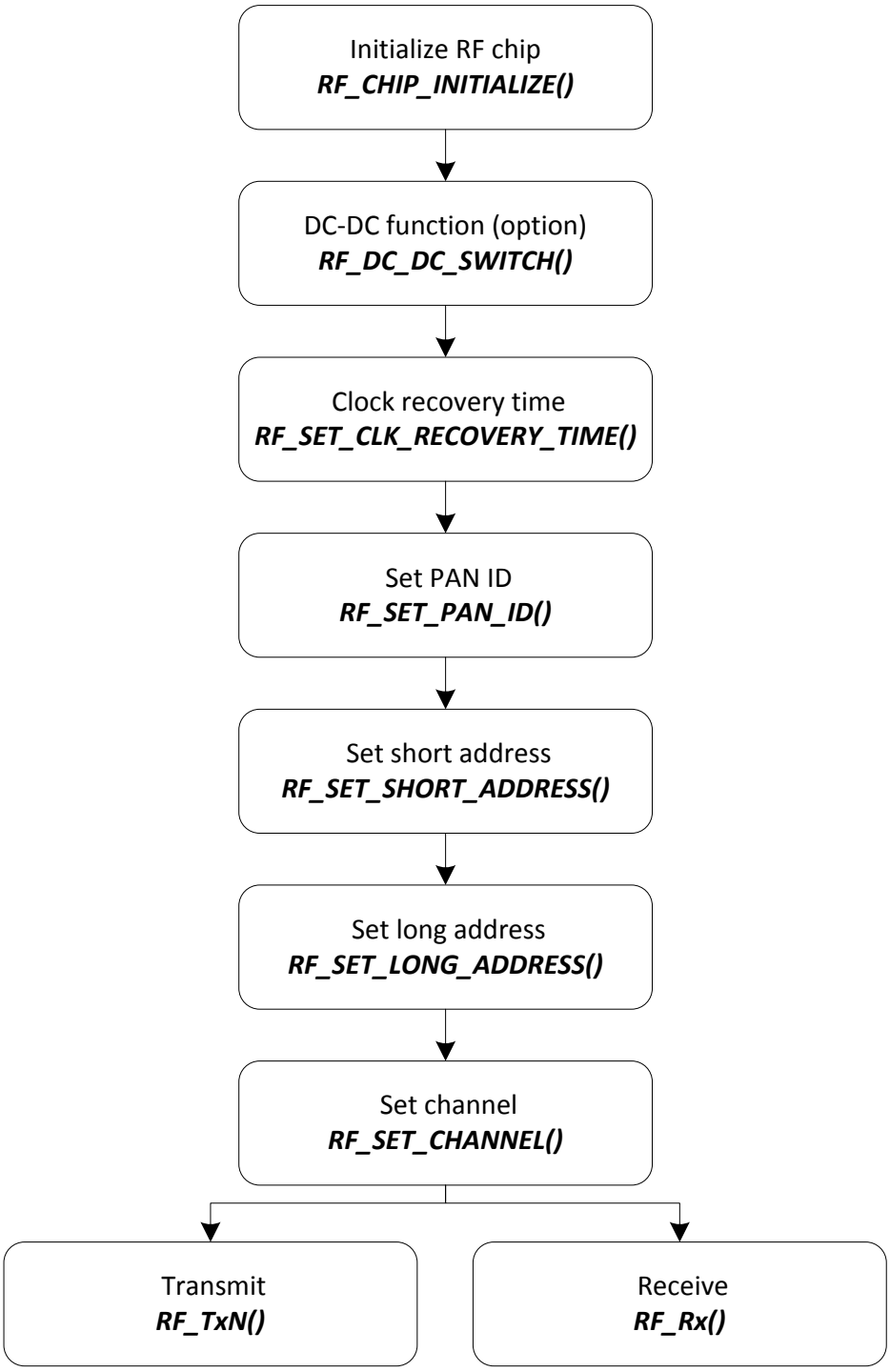
uint32;	unsigned 32bits variable
uint16;	unsigned 16bits variable
uint8;	unsigned 8bits variable
int64;	signed 64bits variable
int32;	signed 32bits variable
int16;	signed 16bits variable
int8;	signed 8bits variable
bool;	1bit variable

3. SPI Function Proto Type

a. Write short register: **spi_sw**(short register address, value)
b. Read short register: **spi_sr**(short register address)
c. Write long register: **spi_lw**(long register address, value)
d. Read long register: **spi_lr**(long register address)
e. Write long register block: **spi_lw_block**(long register start address, data buffer pointer, block size)
f. Read long register block: **spi_lr_block**(long register start address, data buffer pointer, block size)

4. Function Sample Code

4.1. Basic Function Flow Chart



4.2. Initialization (Datasheet 4.3.1)

Function description:
Initializing UZ2400 with DC-DC off and 250Kbps transmission speed.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_CHIP_INITIALIZE(void)
{
      spi_sw(0x2a,   0x07); //reset register
      do
            spi_sw(0x26, 0x20); //enable SPI sync
      while((spi_sr(0x26)&0x20)!= 0x20); //check SPI sync status
      spi_sw(0x17, 0x08); //fine-tune TX timing
      spi_sw(0x18, 0x94); //fine-tune TX timing
      spi_sw(0x2e, 0x95); //fine-tune TX timing
      spi_sw(0x3e, 0x40); //append RSSI value to received packet
      spi_lw(0x200, 0x03); //RF optimized control
      spi_lw(0x201, 0x02); //RF optimized control
      spi_lw(0x202, 0xe6); //RF optimized control
      spi_lw(0x204, 0x06); //RF optimized control
      spi_lw(0x206, 0x30); //RF optimized control
      spi_lw(0x207, 0xe0); //RF optimized control
      spi_lw(0x208, 0x8c); //RF optimized control
      spi_lw(0x23d, 0x00); //Setting GPIO to output mode
      spi_lw(0x24d, 0x20); //enable IEEE802.15.4-2006 security support
      spi_lw(0x250, 0x07); //for DC-DC off, VDD >= 2.4V
      spi_lw(0x251, 0xc0); //RF optimized control
      spi_lw(0x252, 0x01); //RF optimized control
      spi_lw(0x259, 0x00); //RF optimized control
      spi_lw(0x273, 0x40); //RF optimized control, VDD >= 2.4V
      spi_lw(0x274, 0xc6); //for DC-DC off, VDD >= 2.4V
      spi_lw(0x275, 0x13); //RF optimized control
      spi_lw(0x276, 0x07); //RF optimized control
      spi_sw(0x32, 0x00); //clear all interrupt masks
      spi_sw(0x2a, 0x02); //reset baseband
      spi_sw(0x36, 0x04); //reset RF
      spi_sw(0x36, 0x00);
 }
```

4.3. Clock Recovery Time (Datasheet 4.3.2)

4.3.1.

Function description:
Calculate "wake count" and "wake time" for each power saving mode.

Parameter description:
early_wakeup_time: the early wakeup time, unit:us
xtal_stable_time: crystal stable time, unit:us
clk_div: divisor of sleep clock
power_saving_mode: the power saving mode user wants to use
use_ext_wakeup: use external wakeup mode flag

Return value:
The period of sleep clock source, unit: ns

Function definition:

```
uint32 RF_SET_CLK_RECOVERY_TIME(uint16 xtal_stable_time, uint16 early_wakeup_time, uint8 clk_div,
uint8 power_saving_mode, bool use_ext_wakeup)
{
    uint32 sleep_clk, wake_count, wake_time;

    sleep_clk = RF_CAL_SLEEP_CLK(clk_div); //calculate low speed clock, unit:ns    .

    wake_count = (((uint32)xtal_stable_time*1000)/sleep_clk)+1; //calculate wake count.

    switch(power_saving_mode) //fine tune the wake count for each power saving mode.
    {
        case HALT:
            wake_count = 1;     //system clock will not be closed under HALT mode,
                                //the clock recovery time can be reduced to the shortest value.

            break;
        case STANDBY:
            if(use_ext_wakeup) //external wakep circuit will engage 6 clocks of recovery time, it can be
reduced.
            {
                if(clk_div == 0)
                {
                    if((wake_count-6)>0)
                        wake_count = wake_count-6;
                    else
                        wake_count = 1;
                }
                else if(clk_div == 1)
```

```
                {
                    if((wake_count-3)>0) //(6/2) = 3
                        wake_count = wake_count-3;
                    else
                        wake_count = 1;
                }
                else if(clk_div == 2)
                {
                    if((wake_count-1)>0) //(6/4) = 1.5 = 1
                        wake_count = wake_count-1;
                    else
                        wake_count = 1;
                }
            }
            else
            {
                if(wake_count<1)
                    wake_count = 1;
            }
            break;
        default:
            break;
    }

    spi_sw(0x36, (uint8)((wake_count & 0x0180)>> 4)); //set wake count
    spi_sw(0x35, (uint8)(wake_count & 0x007f));

    wake_time = (((uint32)early_wakeup_time*1000)/sleep_clk)+1; //calculate wake time

    if(wake_time <= wake_count) //wake time must be larger than wake count
        wake_time = wake_count+1;

    spi_lw(0x223, (uint8)((wake_time >> 8)&0x07)); //set wake time
    spi_lw(0x222, (uint8)wake_time);

    return sleep_clk;
}
```

4.3.2.

Function description:
Calibrate sleep clock source with selected clock divisor and return the period of clock source.

Parameter description:
clk_div: divisor of sleep clock

Return value:
The period of clock source, unit: ns

Function definition:

```
uint32 RF_CAL_SLEEP_CLK(uint8 clk_div)
{
	uint32 cal_cnt = 0, slp_clk;

	spi_lw(0x220 , clk_div & 0x1f); //set clock div
	spi_lw(0x20b, 0x10); //start calibrate sleep clock

	while(!(spi_lr(0x20b) & 0x80)); //wait calibration ready

	slp_clk = spi_lr(0x20b)& 0xf; //get calibrated value
	cal_cnt |=   slp_clk << 16;
	slp_clk = spi_lr(0x20a);
	cal_cnt |= slp_clk << 8;
	cal_cnt |= spi_lr(0x209);

	slp_clk = ((cal_cnt*125)/32); //calibrate clocl period, unit:ns (cal_cnt * (62.5 * 2)) /(16 * 2)

	return slp_clk;

}
```

4.4. Change Channel Procedure (Datasheet 4.3.3)

Function description:
Use this function to change channel between 2405MHz ~ 2480MHz.

Parameter description:
frequency: Channel frequency, range: 05-80. Please fill the multiple values of 5.
delay: The user defined delay counter value. When change channel, UZ2400 will need delay time.
The length of delay depends on the transmission rate: 250K: 550us, 1M: 300us, 2M: 250us.
This counter value depends on MCU system clock or instruction period.

Return value:
0: set channel unsuccessfully, frequency is out of range
1: set channel successfully

Function definition:

```
bool RF_SET_CHANNEL(uint8 frequency, uint16 delay)
{
	uint8 val;

	if (frequency > 80 || frequency < 05) //check channel range
		return 0; //set channel unsuccessfully, frequency is out of range

	val = spi_sr(0x26); //save register value

	spi_sw(0x26, val|0x2); //turn on TX clock

	spi_lw(0x200, (((frequency/5)-1) << 4)|0x03); //set channel

	spi_sw(0x36, 0x04); //reset RF
	spi_sw(0x36, 0x00);

	while(delay--); //250K: 550us, 1M: 300us, 2M: 250us

	spi_sw(0x26, val); //turn off TX clock, restore original value

	return 1; //set channel successfully
}
```

4.5. External Power Amplifier Configuration (Datasheet 4.3.5)

4.5.1.

Function description:
Enable external PA automatically control.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_AUTO_EXT_PA_CTRL_ON(void)
{
    spi_lw(0x22f, (spi_lr(0x22f)&~0x7)|0x1); //enable external PA automatically control
}
```

4.5.2.

Function description:
Disable external PA automatically control.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_AUTO_EXT_PA_CTRL_OFF(void)
{
    spi_lw(0x22f, spi_lr(0x22f)&~0x7); //disable external PA automatically control
}
```

4.6. Turbo Mode Configuration (Datasheet 4.3.5)

4.6.1.

Function description:
Set transmission speed to 250Kbps.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_NORMAL_SPEED(void)
{
	spi_lw(0x206, 0x30); //RF optimized control
	spi_lw(0x207, spi_lr(0x207)&~0x10); //RF optimized control
	spi_sw(0x38, 0x80); //select 250kbps mode
	spi_sw(0x2a, 0x02); //baseband reset
}
```

4.6.2.

Function description:
Set transmission speed to 1Mbps.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_TURBO_SPEED_1M(void)
{
	spi_lw(0x206, 0x70); //RF optimized control
	spi_lw(0x207, spi_lr(0x207)|0x10); //RF optimized control
	spi_sw(0x38, 0x81); //select 1Mbps mode
	spi_sw(0x2a, 0x02); //reset baseband
}
```

4.6.3.

Function description:
Set transmission speed to 2Mbps.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_TURBO_SPEED_2M(void)
{
	spi_lw(0x206, 0x70); //RF optimized control
	spi_lw(0x207, spi_lr(0x207)&~0x10); //RF optimized control
	spi_sw(0x38, 0x83); //select 2Mbps mode
	spi_sw(0x2a, 0x02); //reset baseband
}
```

4.7. Transmit Packet Using Normal FIFO (Datasheet 4.4.1)

Function description:
Transmit packet using TX normal FIFO.

Parameter description:
*tx_data: the tx data which user wants to transmit
tx_data_len: the length of the tx data
ack_req: acknowledgement require flag

Return value:
1: successfully
0: unsuccessfully, or user data length is larger than 125 bytes

Function definition:

```
uint8 RF_TxN(uint8 *tx_data, uint8 tx_data_len, bool ack_req)
{
	uint8 val;

	if(tx_data_len > 125) //max data length of UZ2400 is 125
		return 0;

	spi_lw(0x1, tx_data_len); //write tx data length
	spi_lw_block(0x2, tx_data, tx_data_len); //load data into tx normal fifo
	val = spi_sr(0x1b);

	if(ack_req) //If need wait Ack?
		val = (val & ~0x02)|0x05;
	else
		val = (val & ~0x06)|0x01;

	spi_sw(0x1b, val); //tirgger TxN

	while(RF_IntIn.TxN == RF_IntProc.TxN); //wait TxN interrupt
	RF_IntProc.TxN ^=1;//claer flag

	if(!(spi_sr(0x24)&0x01)) //check TxN result
		return 1;

	return 0;
}
```

4.8. Transmit Packet in GTS FIFO (Datasheet 4.4.1)

4.8.1.

Function description:
Transmit packet using TX GTS1 FIFO.

Parameter description:
***tx_data**: the tx data which user wants to transmit
tx_data_len: the length of the tx data
ack_req: Acknowledgement require flag
retry: Tx retransmission times, range is 0~3
slot: GTS slot number, range 1~7

Return value:
1: successfully
0: unsuccessfully, or user data length is larger than 125 bytes

Function definition:

```
uint8 RF_TxG1(uint8 *tx_data, uint8 tx_data_len, bool ack_req, uint8 retry, uint8 slot)
{
	uint8 val = 0;

	if(tx_data_len > 125) //Max Data Length of UZ2400 is 125
		return 0;

	spi_lw(0x101, tx_data_len); //load tx data length
	spi_lw_block(0x102, tx_data, tx_data_len); //load data into tx normal fifo

	val = ((retry&0x3)<<6)|((slot&0x7)<<3); //setup retry and slot number

	if(ack_req) //If need wait Ack?
		val |= 0x05;
	else
		val |= 0x01;

	spi_sw(0x1c, val); //trigger

	while(RF_IntIn.TxG1 == RF_IntProc.TxG1); //wait TxG1 interrupt
	RF_IntProc.TxG1 ^=1; //clear flag

	if(!(spi_sr(0x24)& 0x02)) //check TxG1 result
		return 1;

	return 0;
}
```

4.8.2.

Function description:
Transmit packet using TX GTS2 FIFO.

Parameter description:
*tx_data: the tx data which user wants to transmit
tx_data_len: the length of the tx data
ack_req: Acknowledgement require flag
retry: Tx retransmission times, range is 0~3
slot: GTS slot number, range 1~7

Return value:
1: successfully
0: unsuccessfully, or user data length is larger than 125 bytes

Function definition:

```
uint8 RF_TxG2(uint8 *tx_data, uint8 tx_data_len, bool ack_req, uint8 retry, uint8 slot)
{
	uint8 val = 0;

	if(tx_data_len > 125) //Max Data Length of UZ2400 is 125
		return DATA_TOO_LONG;

	spi_lw(0x181, tx_data_len); //load tx data length
	spi_lw_block(0x182, tx_data, tx_data_len); //load data into tx normal fifo

	val = ((retry&0x3)<<6)|((slot&0x7)<<3); //setup retry and slot number

	if(ack_req) //If need wait Ack?
		val |= 0x05;
	else
		val |= 0x01;

	spi_sw(0x1d, val); //trigger

	while(RF_IntIn.TxG2 == RF_IntProc.TxG2); //wait TxG2 interrupt
	RF_IntProc.TxG2 ^=1; //clear flag

	if(!(spi_sr(0x24)& 0x04)) //check TxG2 result
		return 1;

	return 0;
}
```

4.9. Transmit Packet with Security Encryption (Datasheet 4.4.3)

4.9.1.

Function description:
Transmit packet using TX normal FIFO with security encryption.

Parameter description:
* tx_frm: frame pointer.
frm_len: frame length.
ack_req: acknowledgement require flag.
non_enc_len: Non-encrypt data length, this length means byte [0] to byte [non_enc_len] will not be encrypted.
Note: when using MIC mode, the frm_len and non_enc_len must be the same.
sec_mode: security mode value. Please refer to Table 1.

security mode	mode value	maximum length
ENC	0x1	125 bytes
ENC_MIC 128	0x2	109 bytes
ENC_MIC 64	0x3	117 bytes
ENC_MIC 32	0x4	121 bytes
MIC 128	0x2	109 bytes
MIC 64	0x3	117 bytes
MIC 32	0x4	121 bytes

Table 1.

*sec_key: 16 bytes security key pointer.
nonce: nonce is made up with source 64bits address and aux security information.
Please refer to 7.6.3.2 of IEEE 802.15.4 2006.

Return value:
1: successfully
0: security mode is '0' or security mode is larger than '4', or TX procedure has failed.

Function definition:
uint8 RF_SECURE_TxN(uint8 *tx_frm, uint8 frm_len, bool ack_req, uint8 non_enc_len, uint8 sec_mode, uint8 *sec_key, uint8 *nonce)
{
```
    uint8 val;

    if(sec_mode == 0 || sec_mode >4)
        return 0;

    spi_lw(0x0, non_enc_len); //load the length which user doesn't want to encrypt
```

```
	spi_lw(0x1, frm_len); //load data length
	spi_lw_block(0x2, tx_frm, frm_len); //load data
	spi_lw_block(0x240, nonce, 13); //load nonce
	spi_lw_block(0x280, sec_key, 16); //load security key
	spi_sw(0x2c, (spi_sr(0x2c) & ~0x07) |(sec_mode & 0x07)); //Fill encryption mode for TxN FIFO

	val = spi_sr(0x1b);

	if(ack_req) //If need wait Ack?
		val |= 0x07;
	else
		val = (val&~0x04)|0x03;

	spi_sw(0x1b, val); //trigger TxN

	while(RF_IntIn.TxN == RF_IntProc.TxN); //wait TxN interrupt
	RF_IntProc.TxN ^=1;//clear flag

	if(!(spi_sr(0x24)&0x01)) //check TxN result
		return 1;

	return 0;
}
```

4.9.2.

Function description:
Transmit packet using TX GTS1 FIFO with security encryption.

Parameter description:
*tx_frm: user data pointer
frm_len: user data length
ack_req: ACKnowledgement require flag
retry: Tx retransmission times, range is 0~3
slot: GTS slot number, range 1~7
non_enc_len: Non-encrypt data length, this length means byte [0] to byte [non_enc_len] will not be encrypted.
sec_mode: security mode. Please refer to Table 1
*sec_key: 16 bytes security key pointer.
nonce: nonce is made up with source 64bits address and aux security information. Please refer to 7.6.3.2 of IEEE 802.15.4 2006

Return value:
1: successfully
0: security mode is '0' or security mode is larger than '4', or TX procedure has failed.

Function definition:
```
uint8 RF_SECURE_TxG1(uint8 *tx_frm, uint8 frm_len, bool ack_req, uint8 retry, uint8 slot, uint8 non_enc_len,
uint8 sec_mode, uint8 *sec_key, uint8 *nonce)
{
	uint8 val = 0;

	if(sec_mode == 0 || sec_mode >4)
		return 0;

	spi_lw(0x100, non_enc_len); //load the length which user doesn't want to encrypt
	spi_lw(0x101, frm_len); //load data length
	spi_lw_block(0x102, tx_frm, frm_len); //load data
	spi_lw_block(0x240, nonce, 13); //load nonce
	spi_lw_block(0x290, sec_key, 16); //load security key
	spi_sw(0x37, (spi_sr(0x37) & ~0x07) |(sec_mode & 0x07)); //Fill encryption mode for TxG1 FIFO

	val = ((retry&0x3)<<6)|((slot&0x7)<<3); //read value and set retry

	if(ack_req) //If need wait Ack?
		val |= 0x07;
	else
		val |= 0x03;

	spi_sw(0x1c, val); //trigger TxG1
```

```
    while(RF_IntIn.TxG1 == RF_IntProc.TxG1); //wait TxG1 interrupt
    RF_IntProc.TxG1 ^=1; //clear flag

    if(!(spi_sr(0x24)& 0x02)) //check TxG1 result
        return 1;

    return 0;
}
```

4.9.3.

Function description:

Transmit packet using TX GTS2 FIFO with security encryption.

Parameter description:

*tx_frm: user data pointer
frm_len: user data length
ack_req: ACKnowledgement require flag
retry: Tx retransmission times, range is 0~3
slot: GTS slot number, range 1~7
non_enc_len: Non-encrypt data length, this length means byte [0] to byte [non_enc_len] will not be encrypted.
sec_mode: security mode. Please refer to Table 1
*sec_key: 16 bytes security key pointer.
nonce: nonce is made up with source 64bits address and aux security information. Please refer to 7.6.3.2 of IEEE 802.15.4 2006

Return value:

1: successfully

0: security mode is '0' or security mode is larger than '4', or TX procedure has failed.

Function definition:

```
uint8 RF_SECURE_TxG2(uint8 *tx_frm, uint8 frm_len, bool ack_req, uint8 retry, uint8 slot, uint8 non_enc_len,
uint8 sec_mode, uint8 *sec_key, uint8 *nonce)
{
	uint8 val = 0;

	if(sec_mode == 0 || sec_mode >4)
		return 0;

	spi_lw(0x180, non_enc_len); //load the length which user doesn't want to encrypt
	spi_lw(0x181, frm_len); //load data length
	spi_lw_block(0x182, tx_frm, frm_len); //load data
	spi_lw_block(0x240, nonce, 13); //load nonce
	spi_lw_block(0x2A0, sec_key, 16); //load security key
	spi_sw(0x37, (spi_sr(0x37) & ~0x38)|((sec_mode & 0x07)<<3)); //Fill encryption mode for TxG2 FIFO

	val = ((retry&0x3)<<6)|((slot&0x7)<<3); //read value and clear retry

	if(ack_req) //If need wait Ack?
		val |= 0x07;
	else
		val |= 0x03;

	spi_sw(0x1d, val); //trigger TxG2

	if(!(spi_sr(0x24)& 0x04)) //check TxG2 result
		return 1;

	return 0;
}
```

4.10. Transmit Packet in Normal FIFO with CCA/ED Mode or Combination of CS and ED Modes (Datasheet 4.4.4)

Function description:

Transmit packet using TX normal FIFO.

Parameter description:

*tx_data: user data pointer

tx_data_len: user data length

ack_req: acknowledgement require flag

Return value:

1: successfully

0: unsuccessfully, or user data length is larger than 125 bytes

Function definition:

```
uint8 RF_TxN(uint8 *tx_data, uint8 tx_data_len, bool ack_req)
{
	uint8 val;

	if(tx_data_len > 125) //max data length of UZ2400 is 125
		return 0;

	spi_lw(0x1, tx_data_len); //write tx data length
	spi_lw_block(0x2, tx_data, tx_data_len); //load data into tx normal fifo
	val = spi_sr(0x1b);

	if(ack_req) //If need wait Ack?
		val = (val & ~0x02)|0x05;
	else
		val = (val & ~0x06)|0x01;

	spi_sw((spi_sr(0x3a)&~0xc0)|0x80); // Set CCA to ED mode
	// spi_sw((spi_sr(0x3a)&~0xc0)|0xc0); // combination of CS and ED modes

	spi_sw(0x1b, val); //tirgger TxN

	while(RF_IntIn.TxN == RF_IntProc.TxN); //wait TxN interrupt
	RF_IntProc.TxN ^=1;//claer flag

	spi_sw((spi_sr(0x3a)&~0xc0)|0x40); // Set CCA back to CS mode

	if(!(spi_sr(0x24)&0x01)) //check TxN result
		return 1;

	return 0;
}
```

4.11. Receive Packet in RXFIFO (Datasheet 4.5.1)

Function description:
Check RX interrupt and read received packet from RX FIFO.

Parameter description:
*buffer: user data buffer pointer

Return value:
Received packet length, '0' means that there is no received packet.

Function definition:

```
uint8 RF_Rx(uint8 *buffer)
{
	uint8 len;

	if(RF_IntIn.Rx != RF_IntProc.Rx) //check rx interrupt
	{
		RF_IntProc.Rx ^=1; //clear flag
		len = spi_lr(0x300); //read received packet length
		spi_lr_block(0x301, buffer, len); //read received packet
		return len-2; //decrease CRC length
	}

	return 0;
}
```

4.12. Receive Packet with Security Decryption (Datasheet 4.5.2)

Function description:
Check and read received packet from RX FIFO with security decryption.

Parameter description:
* rx_buf: receive buffer pointer.
sec_mode: security mode value, please refer to table 1.
*sec_key: 16 bytes security key pointer.
*src_ladr: long address pointer of source (TX) device.
non_enc_len: Non-encrypt data length.
nonce: nonce is made up with the 64bits address of source device and aux security information.
Please refer to 7.6.3.2 of IEEE 802.15.4 2006.

Return value:
1-125: Received packet length
0: means that there is no received packet.

Function definition:

```
uint8 RF_SECURE_Rx(uint8 *rx_buf, uint8 sec_mode, uint8 *sec_key, uint8 *src_ladr, uint8 non_enc_len,
uint8 *nonce)
{
    uint8 len;

    if(RF_IntIn.Sec != RF_IntProc.Sec) //check security interrupt
    {
        RF_IntProc.Sec ^=1; //clear flag

        //
        // Here user can accord the received header length and frame length to
        // read AUX header from RXFIFO and process security information.
        //

        hdr_len = spi_lr(0x21e); //read received header length
                                 //(this value doesn't include AUX header length)
        len = spi_lr(0x300);    //read received total frame length

        if(sec_mode == 0)
        {
            spi_sw(0x2c, spi_sr(0x2c)|0x80); //ignore rx security decryption
            goto READ_FIFO;
        }
        else
        {
            spi_lw(0x21e, non_enc_len);
```

```
            }

            if((spi_lr(0x212) & 0x3) == 0x2)//check source address mode of received packet
                  spi_lw_block(0x213, src_ladr, 8); //load source device 64bits address

            spi_lw_block(0x240, nonce, 13); //load nonce
            spi_lw_block(0x2b0, sec_key, 16);//load secure key for RxFIFO
            spi_sw(0x2c, (spi_sr(0x2c)& ~0x38)|((sec_mode & 0x7) <<3)|0x40); //fill decryption mode,
trigger security process

READ_FIFO:

            while(RF_IntIn.Rx == RF_IntProc.Rx); //wait Rx interrupt
            RF_IntProc.Rx ^=1; //clear flag

            if(spi_sr(0x30) & 0x04) //check decryption error status
            {
                  spi_sw(0x0d, spi_sr(0x0d)|0x1); //Rx flush
                  return 0;
            }

            len = spi_lr(0x300); //read received packet length
            spi_lr_block(0x301, rx_buf, len); //read received packet

            if(sec_mode == 2) //decrease mic length
                  len = len-16;
            else if(sec_mode == 3)
                  len = len-8;
            else if(sec_mode == 4)
                  len = len-4;

            return len-2; //decrease CRC length
      }

      return 0;
}
```

4.13. Beacon Mode Setting (Datasheet 4.6.1)

4.13.1.

Function description:
Configure a device as a pan coordinator, enable slotted-netwrok.

Parameter description:
sleep_clk: the period of sleep clock source
bo: beacon order, range 1~14.
so: superframe order, range 1~14.
*bcn_frm: beacon frame pointer.
frm_len: the length of beacon frame

Return value:
1: success
0: so, bo value range error, or the period of sleep clock is too short, cause the timer counter to overflow.

Function definition:

```
uint8 RF_ACT_COORDINATOR_IN_SLOTTED_MODE(uint32 sleep_clk, uint8 bo, uint8 so, uint8 *bcn_frm,
uint8 frm_len)
{
	uint32 tmp;
	int64 beacon_interval = 15360000L; //unit: ns, 15360000 ns = aBaseSuperframeDuration(us) * 1000;
					//aBaseSuperframeDuration = 60(symbols) * 16(us)(symbol time) * 16(slots)

	if((so > bo)||(bo >14)||(so > 14)) //beacon order shall be equal to or lager than super frame order
		return 0;

	spi_sw(0x10, 0xff); //stop slotted-network

	beacon_interval = beacon_interval*(1 << bo); //calculate beacon interval

	//
	// calculate and configure slotted-mode timer
	//

	tmp = (uint32)((beacon_interval-(sleep_clk*4))/sleep_clk); //calculate main counter

	if(tmp < 0x4000000) //check main counter overflow
	{
		spi_lw(0x226, (uint8)(tmp & 0x000000ff)); //set main counter
		spi_lw(0x227, (uint8)((tmp & 0x0000ff00)>> 8));
		spi_lw(0x228, (uint8)((tmp & 0x00ff0000)>> 16));
		spi_lw(0x229, (uint8)((tmp & 0x03000000)>> 24));
```

```
            beacon_interval = beacon_interval - ((int64)sleep_clk*(int64)tmp);
            tmp = (uint32)(beacon_interval/62.5); //calculate remain counter

            if(tmp < 0x10000) //check remain counter overflow
            {
                spi_lw(0x225, (uint8)((tmp & 0x0000ff00)>>8)); //set remain counter
                spi_lw(0x224, (uint8)(tmp & 0x000000ff));

                //
                // configure device as a pan coordinator
                //

                spi_sw(0x11, spi_sr(0x11)|0x20); //enable slotted mode
                spi_sw(0x00, spi_sr(0x00)|0x08); //configure as the PAN coordinator, coordinator
                spi_sw(0x22, (spi_sr(0x22)&~0x3f)|0x03);
                spi_sw(0x25, spi_sr(0x25)|0x80); //mask the beacon interrupt mask.
                spi_sw(0x26, spi_sr(0x26)|0x08); //enable gts fifo clock
                RF_ALLOCATE_END_SLOT(15, 0, 0, 0, 0, 0, 0); //allocate all slots to cap

                spi_lw(0x81, frm_len); //load beacon frame length
                spi_lw_block(0x82, bcn_frm, frm_len); //load beacon frame

                spi_sw(0x10, ((bo & 0xf) << 4)|(so & 0xf)); //set bo, so, the slotted-network will start
automatically

                return 1;
            }
        }

        return 0;
}
```

4.13.2.

Function description:
Configure a device as a device of slotted-network.

Parameter description:
sleep_clk: the period of sleep clock source
bo: beacon order, range 1~14.
so: superframe order, range 1~14.
*ac_adr,: device address of the coordinator which you want to join.
ac_adr_len: address length, value = 2 is 16bits, value = 8 is 64bits.

Return value:
1: success
0: so, bo value range error, or the period of sleep clock is too short, cause the timer counter to overflow.

Function definition:

```
uint8 RF_ACT_DEVICE_IN_SLOTTED_MODE(uint32 sleep_clk, uint8 bo, uint8 so, uint8* ac_adr, uint8
ac_adr_len)
{
	uint32 tmp;
	int64 inactive_interval = 15360000L; //unit: ns, 15360000 ns = aBaseSuperframeDuration(us) * 1000;
                                                    //aBaseSuperframeDuration = 60(symbols) *
16(us)(symbol time) * 16(slots)

	if((so > bo)||(bo >14)||(so > 14)) //beacon order shall be equal to or lager than super frame order
		return 0;

	spi_sw(0x10, 0xff); //stop slotted-network

	inactive_interval = inactive_interval*((1 << bo)-(1 << so)); //calculate inactive interval

	//
	// calculate and configure slotted-mode timer
	//

	tmp = (uint32)((inactive_interval-(sleep_clk*4))/sleep_clk); //calculate main counter

	if(tmp < 0x4000000) //check main counter overflow
	{
		spi_lw(0x226, (uint8)(tmp & 0x000000ff)); //set main counter
		spi_lw(0x227, (uint8)((tmp & 0x0000ff00)>> 8));
		spi_lw(0x228, (uint8)((tmp & 0x00ff0000)>> 16));
		spi_lw(0x229, (uint8)((tmp & 0x03000000)>> 24));

		inactive_interval = inactive_interval - ((int64)sleep_clk*(int64)tmp);
```

```
            tmp = (uint32)(inactive_interval/62.5); //calculate remain counter

            if(tmp < 0x10000) //check remain counter overflow
            {
                  spi_lw(0x225, (uint8)((tmp & 0x0000ff00)>>8)); //set remain counter
                  spi_lw(0x224, (uint8)(tmp & 0x000000ff));

                  //
                  // configure device as a slotted device
                  //

                  spi_sw(0x11, spi_sr(0x11)|0x20); //enable slotted mode
                  spi_sw(0x22, (spi_sr(0x22)&~0x3f)|0x03);
                  spi_sw(0x23, 0x15); //timing alignment
                  spi_sw(0x26, spi_sr(0x26)|0x08); //enable gts fifo clock
                  RF_ALLOCATE_END_SLOT(15, 0, 0, 0, 0, 0, 0);   //allocate all slots to cap

                  if(ac_adr_len == 2) //load coordinator address
                        RF_SET_AC_SHORT_ADDRESS(*((uint16 *)ac_adr));
                  else if(ac_adr_len == 8)
                        RF_SET_AC_LONG_ADDRESS(ac_adr);
                  else
                        return 0;

                  spi_sw(0x10, ((bo & 0xf) << 4)|(so & 0xf)); //set bo, so, the slotted-network will start
automatically

                  return 1;
            }
      }

      return 0; //counter overflow
}
```

4.13.3.

Function description:
Set associated coordinator short address for slotted network synchronization.

Parameter description:
sadr: associated coordinator short address.

Return value:
None

Function definition:
```
void RF_SET_AC_SHORT_ADDRESS(uint16 sadr)
{
	spi_lw(0x239, (uint8)((sadr & 0xff00) >> 8)); //msb
	spi_lw(0x238, (uint8)(sadr & 0x00ff)); //lsb
}
```

4.13.4.

Function description:
Set associated coordinator long address for slotted network synchronization.

Parameter description:
*ladr: associated coordinator long address pointer.

Return value:
None

Function definition:
```
void RF_SET_AC_LONG_ADDRESS(uint8 *ladr)
{
	uint8 i;

	for (i=0; i<8; i++)
		spi_lw(0x231+i, *(ladr+i)); //set value , LSB first, MSB last
}
```

4.14. Beacon Mode GTS Setting (Datasheet 4.6.2)

Function description:
Set the end slot number of the GTS duration.

Parameter description:
cap: the end slot number of cap duration
gts1: the end slot number of gts1 duration
gts2: the end slot number of gts2 duration
gts3: the end slot number of gts3 duration
gts4: the end slot number of gts4 duration
gts5: the end slot number of gts5 duration
gts6: the end slot number of gts6 duration

Return value:
None

Function definition:

```
void RF_ALLOCATE_END_SLOT(uint8 cap, uint8 gts1, uint8 gts2, uint8 gts3, uint8 gts4, uint8 gts5, uint8 gts6)
{
	spi_sw(0x13, ((gts1&0xf)<<4)|(cap&0xf));
	spi_sw(0x1e, ((gts3&0xf)<<4)|(gts2&0xf));
	spi_sw(0x1f, ((gts5&0xf)<<4)|(gts4&0xf));
	spi_sw(0x20, gts6&0xf);
}
```

4.15. Wakeup Operations (Datasheet 4.7.1)

4.15.1.

Function description:
Enable external wake-up pin and set the polarity of wake-up signal.

Parameter description:
trigger_polarity: 1, active high; 0, active low

Return value:
None

Function definition:

```
void RF_ENABLE_EXT_WAKEUP_PIN(bool trigger_polarity)
{
    if(trigger_polarity)
        spi_sw(0x0d, (spi_sr(0x0d)&0x08)|0x40); //enable external wake up, active high
    else
        spi_sw(0x0d, spi_sr(0x0d)&0x08); //enable external wake up, active low
}
```

4.15.2.

Function description:
External wake-up.

Parameter description:
trigger_polarity: 1, active high; 0, active low

Return value:
None

Function definition:

```
void RF_EXT_PIN_WAKEUP(bool trigger_polarity)
{
    if(trigger_polarity) //wake up pluse
    {
        UBEC_WAKE_PORT = 1; //high
        UBEC_WAKE_PORT = 0; //low
    }
    else
    {
        UBEC_WAKE_PORT = 0; //low
        UBEC_WAKE_PORT = 1; //high
    }
```

```
    while(RF_IntIn.Wakeup == RF_IntProc.Wakeup); //wait wake up interrupt
    RF_IntProc.Wakeup ^=1;

    do
        spi_sw(0x26, spi_sr(0x26)|0x20); //enable SPI sync
    while((spi_sr(0x26)&0x20)!= 0x20);

    if(DCDC_FLAG) //if DC-DC enabled is needed?
        spi_lw (0x250, spi_lr(0x250)|0x10);    //DC-DC on

}
```

4.15.3.

Function description:
Register wake-up.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_REGISTER_WAKEUP(void)
{
    spi_sw(0x22, 0xc0); //register wake up

    while(RF_IntIn.Wakeup == RF_IntProc.Wakeup); //wait wake up interrupt
    RF_IntProc.Wakeup ^=1;

    do
        spi_sw(0x26, spi_sr(0x26)|0x20); //enable SPI sync
    while((spi_sr(0x26)&0x20)!= 0x20);

    if(DCDC_FLAG) //if DC-DC enabled is needed?
        spi_lw (0x250, spi_lr(0x250)|0x10);    //DC-DC on

    spi_sw(0x22, 0x0);
}
```

4.16. Power Saving Operations (Datasheet 4.7.2)

4.16.1.

Function description:
Enable clock output on CLK_OUT pin and pre configure HALT mode setting.

Parameter description:
output_clock: output clock selection, LREG0x207[7:5].

Return value:
None

Function definition:

```
void RF_HALT_PRE_CONFIG(uint8 output_clock)
{
    spi_lw(0x207, output_clock&0xe0); //set output clock
    spi_lw(0x277, 0x04);
    spi_lw(0x255, 0x20);
}
```

4.16.2.

Function description:
Pre configure STANDBY mode setting.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_STANDBY_PRE_CONFIG(void)
{
    spi_lw(0x255, 0x00);
    spi_lw(0x277, 0x08);
}
```

4.16.3.

Function description:
Pre configure DEEP SLEEP mode setting.

Parameter description:
None

Return value:
None

Function definition:
```
void RF_DEEP_SLEEP_PRE_CONFIG(void)
{
	spi_lw(0x255, 0x00);
	spi_lw(0x277, 0x18);
}
```

4.16.4.

Function description:
Pre configure POWER DOWN mode setting.

Parameter description:
None

Return value:
None

Function definition:
```
void RF_POWER_DOWN_PRE_CONFIG(void)
{
	spi_lw(0x255, 0x00);
	spi_lw(0x277, 0x38);
	spi_lw(0x253, spi_lr(0x253)|0x60);
}
```

4.16.5.

Function description:
Get into power saving mode.

Parameter description:
None

Return value:
None

Function definition:
```
void RF_START_POWER_SAVING(void)
{
	if(DCDC_FLAG) //if DC-DC on
		spi_lw (0x250, spi_lr(0x250)&~0x10); //turn DC-DC off
```

```
    spi_sw(0x26, spi_sr(0x26)&~0x20); //disable SPI sync

    spi_sw(0x35, spi_sr(0x35)|0x80); //put ic into power saving
}
```

4.16.6.

Function description:
Wake-up function for POWER DOWN mode.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_QUIT_POWER_DOWN(bool trigger_polarity)
{
    if(trigger_polarity)
    {
        UBEC_WAKE_PORT = 1;    //wake up pin high
        UBEC_WAKE_PORT = 0;    //wake up pin low
    }
    else
    {
        UBEC_WAKE_PORT = 0;    //wake up pin low
        UBEC_WAKE_PORT = 1;    //wake up pin high
    }

    RF_CHIP_INITIALIZE();
}
```

4.17. Battery Monitors Operations (Datasheet 4.8)

4.17.1.

Function description:
Check current voltage.

Parameter description:
volth: voltage threshold, defined at LREG0x205[7:4].

Return value:
1: voltage is lower than threshold user defined.
0: voltage is higher than threshold user defined.

Function definition:

```
bool RF_VOLTAGE_CHECK(uint8 volth)
{
	spi_lw(0x205, volth&0xf0);		//Set threshold
	spi_lw(0x206, spi_lr(0x206)|0x08);	//Enable voltage monitor
	volth = spi_sr(0x34);
	spi_lw(0x206, spi_lr(0x206)&~0x08); //Disable voltage monitor
	if(volth & 0x20)			//Check voltage low indicator
		return 1; //Low indicator was set, voltage now is lower than the threshold user set

	return 0;   //Low indicator was not set, voltage now is higher than the threshold user set
}
```

4.17.2.

Function description:
DC-DC switch and setting function with voltage threshold for 2.4V.

Parameter description:
None

Return value:
None

Function definition:

```
void RF_DC_DC_SWITCH(void)
{
	uint8 val;

	val = spi_lr(0x253)&0x0f;

	if(RF_VOLTAGE_CHECK(0x70)) //check voltage threshold 2.5V
```

```
    {
        if(DCDC_FLAG) //if DC-DC is working.
        {
            spi_lw(0x250, spi_lr(0x250)&~0x10); //if voltage is lower than 2.5V, DC-DC bypass

            if(val == 0x0) //if power is 0dBm
                spi_lw(0x274, 0xc0);

            spi_lw(0x273, 0x4e);

            DCDC_FLAG = 0;
        }
    }
    else
    {
        if(!DCDC_FLAG) //if DC-DC is not working.
        {
            spi_lw(0x250, spi_lr(0x250)|0x10); //if voltage is higher than 2.4V, DC-DC on

            if(val == 0x0)
                spi_lw(0x274, 0xc4);

            spi_lw(0x273, 0x40);

            DCDC_FLAG = 1;
        }
    }
}
```

4.18. Upper-Layer-Cipher Encryption (Datasheet 4.9.1)

Function description:

Using TX normal FIFO to do upper cipher encryption

Parameter description:

*raw_data: input raw data pointer.

raw_data_len: input raw data length.

non_enc_len: non-encrypt data length, this length means the byte[0] to the byte[non_enc_len] will not be encrypted.

sec_mode: security mode.

*sec_key: security key pointer, security key is 16 bytes.

nonce: security nonce pointer, nonce is 13 bytes

Return value:

0: security mode error or encryption error.

others: encrypted data length.

Function definition:

```
uint8 RF_UPPER_CIPHER(uint8 *raw_data, uint8 raw_data_len, uint8 non_enc_len, uint8 sec_mode, uint8 *sec_key, uint8 *nonce)
{
	uint8 len;

	if(sec_mode == 0 || sec_mode >4)
		return 0;

	spi_lw(0x0, non_enc_len); //load non encrypt data length into TXNFIFO
	spi_lw(0x1, raw_data_len); //write raw data length
	spi_lw_block(0x2, raw_data, raw_data_len); //load raw data into TX FIFO
	spi_lw_block(0x240, nonce, 13); //load nonce
	spi_lw_block(0x280, sec_key, 16); //loadl security key
	spi_sw(0x2c, (spi_sr(0x2c) & ~0x07) |(sec_mode & 0x07)); //set encryption mode
	spi_sw(0x37, spi_sr(0x37)|0x40); //enable upper cipher mode
	spi_sw(0x1b, (spi_sr(0x1b) & ~0x07)|0x03); //Trigger Tx

	while(RF_IntIn.TxN == RF_IntProc.TxN); //wait TxN interrupt
	RF_IntProc.TxN ^=1; //clear flag

	if(spi_sr(0x24) & 0x01) //check status
		return 0;

	len = spi_lr(0x1); //read length
	spi_lr_block(0x2, raw_data, len); //read encrypted data

	return len;
}
```

4.19. Upper-Layer-Cipher Decryption (Datasheet 4.9.2)

Function description:
Using TX normal FIFO to do upper cipher decryption.

Parameter description:
*encrypted_data: encrypted data pointer.
encrypted_data_len: encrypted data length.
non_enc_len: non-encrypted part data length, this length means the byte[0] to
the byte[non_enc_len] were not be encrypted by the function RF_UPPER_CIPHER().
sec_mode: security mode.
*sec_key: security key pointer, security key is 16 bytes.
nonce: security nonce pointer, nonce is 13 bytes.

Return value:
0: security mode error, decryption error or mic check error.
others: decrypted data length.

Function definition:

```
uint8 RF_UPPER_DECIPHER(uint8 *encrypted_data, uint8 encrypted_data_len, uint8 non_enc_len, uint8
sec_mode, uint8 *sec_key, uint8 *nonce)
{
	uint8 len;

	if(sec_mode == 0 || sec_mode >4)
		return 0;

	if(sec_mode > 0x1)
		encrypted_data_len += 2;

	spi_lw(0x0, non_enc_len); //write non encrypted part data length
	spi_lw(0x1, encrypted_data_len); //write total data length
	spi_lw_block(0x2, encrypted_data, encrypted_data_len); //load data into TX FIFO
	spi_lw_block(0x240, nonce, 13); //load nonce
	spi_lw_block(0x280, sec_key, 16); //load security key
	spi_sw(0x2c, (spi_sr(0x2c) & ~0x07) |(sec_mode & 0x07)); //set encryption mode
	spi_sw(0x37, spi_sr(0x37)|0x80); //enable upper cipher mode
	spi_sw(0x1b, (spi_sr(0x1b) & ~0x07)|0x03); //Trigger Tx

	while(RF_IntIn.TxN == RF_IntProc.TxN); //wait interrupt
	RF_IntProc.TxN ^=1;

	if(spi_sr(0x30) & 0x20) //check mic error
	{
		spi_sw(0x30, 0x20);
```

```
        return 0;
    }

    len = spi_lr(0x1);

    if((sec_mode == AES_ENC_MIC_128)||(sec_mode == AES_MIC_128))
        len = len-16-2;
    else if((sec_mode == AES_ENC_MIC_64)||(sec_mode == AES_MIC_64))
        len = len-8-2;
    else if((sec_mode == AES_ENC_MIC_32)||(sec_mode == AES_MIC_32))
        len = len-4-2;

    spi_lr_block(0x2, encrypted_data, len);

    return len;
}
```

Revision History

Revision	Date	Description of Change
0.0	2009/11/30	Initialization

Contact UBEC:

Headquarters
Address: 6-1, No. 192, Dongguang Rd., Hsinchu, 300 Taiwan
Tel: +886-3-5729898
Fax: +886-3-5718599
Website: http://www.ubec.com.tw

Sales Services
Tel: +886-3-5729898
Fax: +886-3-5718599
E-mail: sales@ubec.com.tw

FAE Services
Tel: +886-3-5729898
Fax: +886-3-5718599
E-mail: fae@ubec.com.tw

DISCLAIMER