AN415

EZRadioPRO® Programming Guide

1. Introduction

This document gives an overview of configuring the EZRadioPRO radios for transmitter, receiver, and transceiver operation via several simple software examples.

The following examples are covered in this programming guide:

- How to use the EZRadioPRO transmitter or transceiver for packet transmission in FIFO mode
- How to use the EZRadioPRO receiver or transceiver for packet reception in FIFO mode
- How to use the EZRadioPRO transceiver for bidirectional packet-based communication
- How to transmit and receive packets with longer than 64 bytes payload using the FIFO

The latest example source code is available on the Silicon Labs website:
www.silabs.com/products/wireless/EZRadioPRO/ or on the WDS CDROM that ships with the evaluation board kits.

2. Hardware Options

The source code is provided for three EZRadioPRO transceiver chips: Si4431 Revision A0, Si4432 Revision V2, and Si443x Revision B1. There are few differences between the radios. The Si443x-B1 is the latest production revision. All the errata items of Si4431-A0 and Si4432-V2 are corrected on this silicon.

- The Si4432-V2 requires some registers to be programmed to values other than their default values. These are not needed for the Si4431-A0.
- The Si4431-A0 and Si443x-B1 have separate registers for setting the Auto-Frequency Calibration limit; however, the Frequency Deviation register is used for this purpose if the Si4432-V2 is used.
- Different modem parameters have to be used for all revisions.
- Invalid preamble timeout is defined differently for the Si4432-V2.

Separate source code examples are provided for all revisions with the differences highlighted in the programming guide.

Note: While only the Si4431-A0, Si4432-V2, and Si443x-B1 devices are mentioned in the document, the transmit or receive software works with the EZRadioPRO standalone transmitters and receivers.

- The Si4431-A0 transmit example code works with the Si4031-A0 without any changes. The code is also applicable to the Si4430-A0 or Si4030- A0, but the center frequency has to be set to the appropriate value.
- The transmit example code provided for the Si4432-V2 works with the Si4032-V2 without any changes.
- The receive example code for the Si4431 works with the Si4330-A0 without any changes.
- The Si443x-B1 transmit example code works with the Si403x-B1 without any changes. The code is also applicable to the Si4430-B1 and Si4030-B1, but the center frequency has to be set to the appropriate value.
- The receiver example code for the Si443x-B1 works with the Si4330-B1 without any changes.

A separate Silicon Labs IDE workspace is provided for each example on the platforms. The name of the Silicon Labs IDE file shows the platform for which the given code is written:

- Workspace filenames containing “SDBC_DK3” are written for the Software Development board.
- Workspace filenames containing “EZLINK” are written for the EZLink platform.

2.1. Antenna Options

The power amplifier and the LNA are not connected to one another inside the EZRadioPRO devices. When using the Si4431 transceivers, the TX and RX pins can be directly connected externally, eliminating the need for an RF switch. When using the highest output power settings of the Si4432, separate transmit and receive pins on the RFIC are connected to an antenna via an SPDT RF switch. The EZRadioPRO devices assist with the control of the RF switches. By routing the *RX State* and *TX State* signals to any two GPIOs, the radio can automatically control the RF switch. The GPIOs control the RF switch to automatically connect the antenna to the receive path or transmit depending on the mode of operation. The GPIOs will disable the RF switch if the radio is not in active mode.

On the Single Antenna with RF Switch Testcard and on the EZLink SIL module, the same RF switch configuration is used: the TX State signal is routed to GPIO1 and the RX State signal is routed to GPIO2.

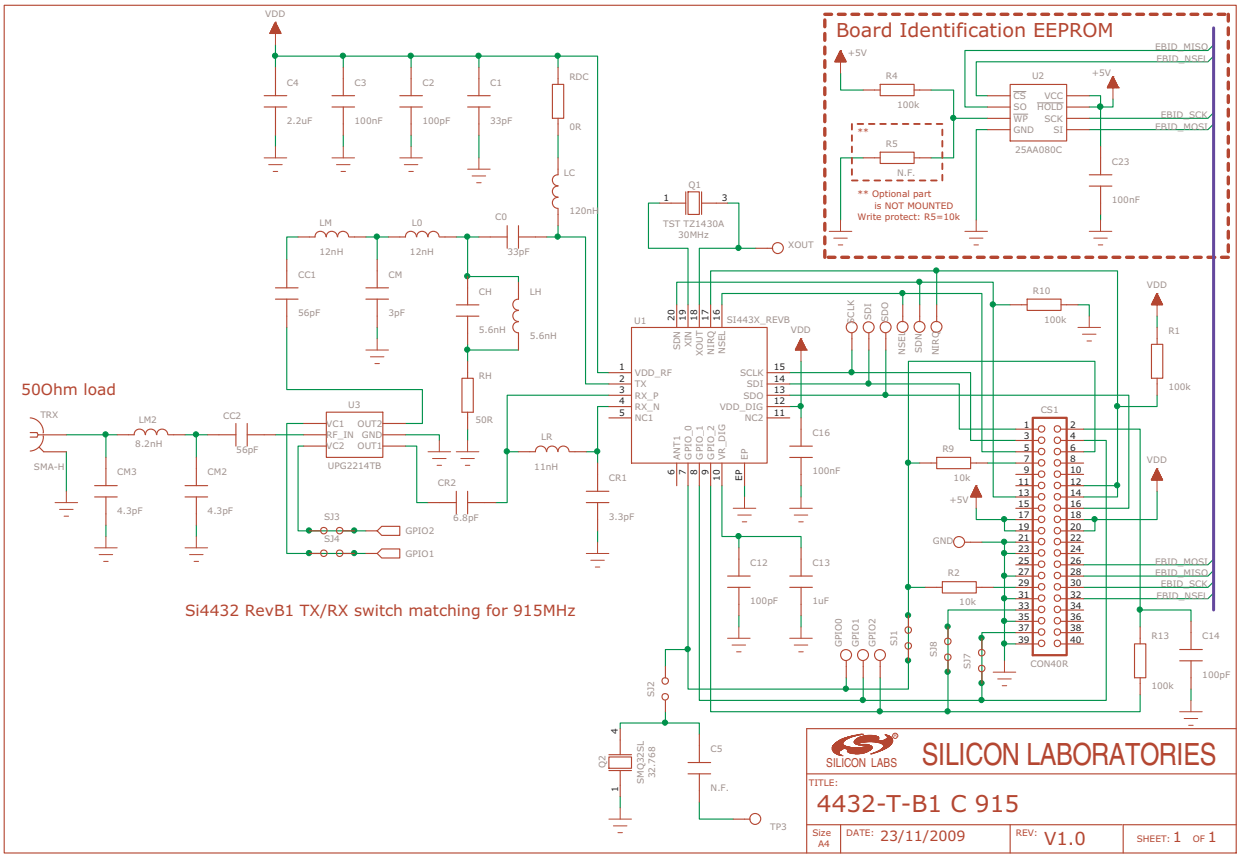


Figure 1. Single Antenna with RF Switch Si4432 Testcard Schematic

If the Si4431-B1 device is used, it is possible to tie together the receive and transmit paths on the PCB directly. It is not necessary to control the RF switch; therefore, all the GPIOs can be used for any other feature.

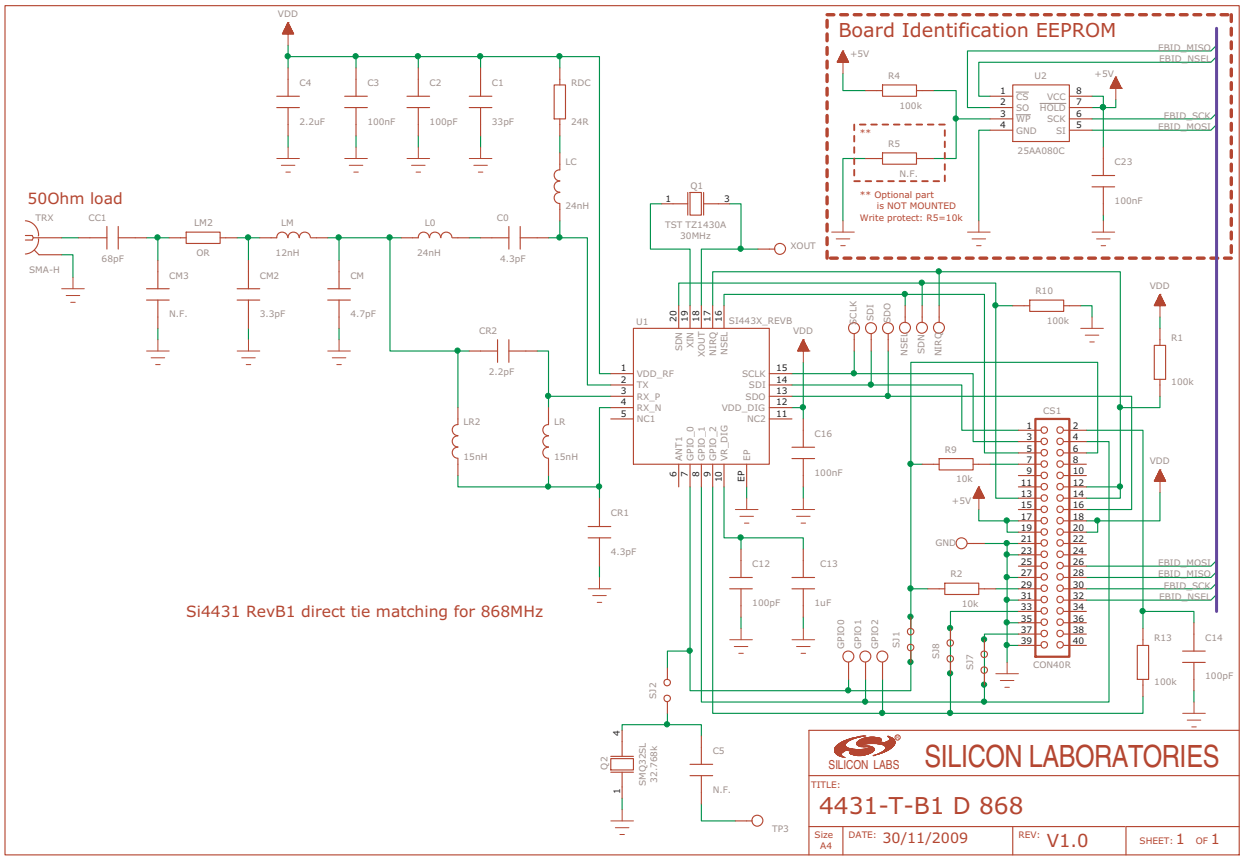
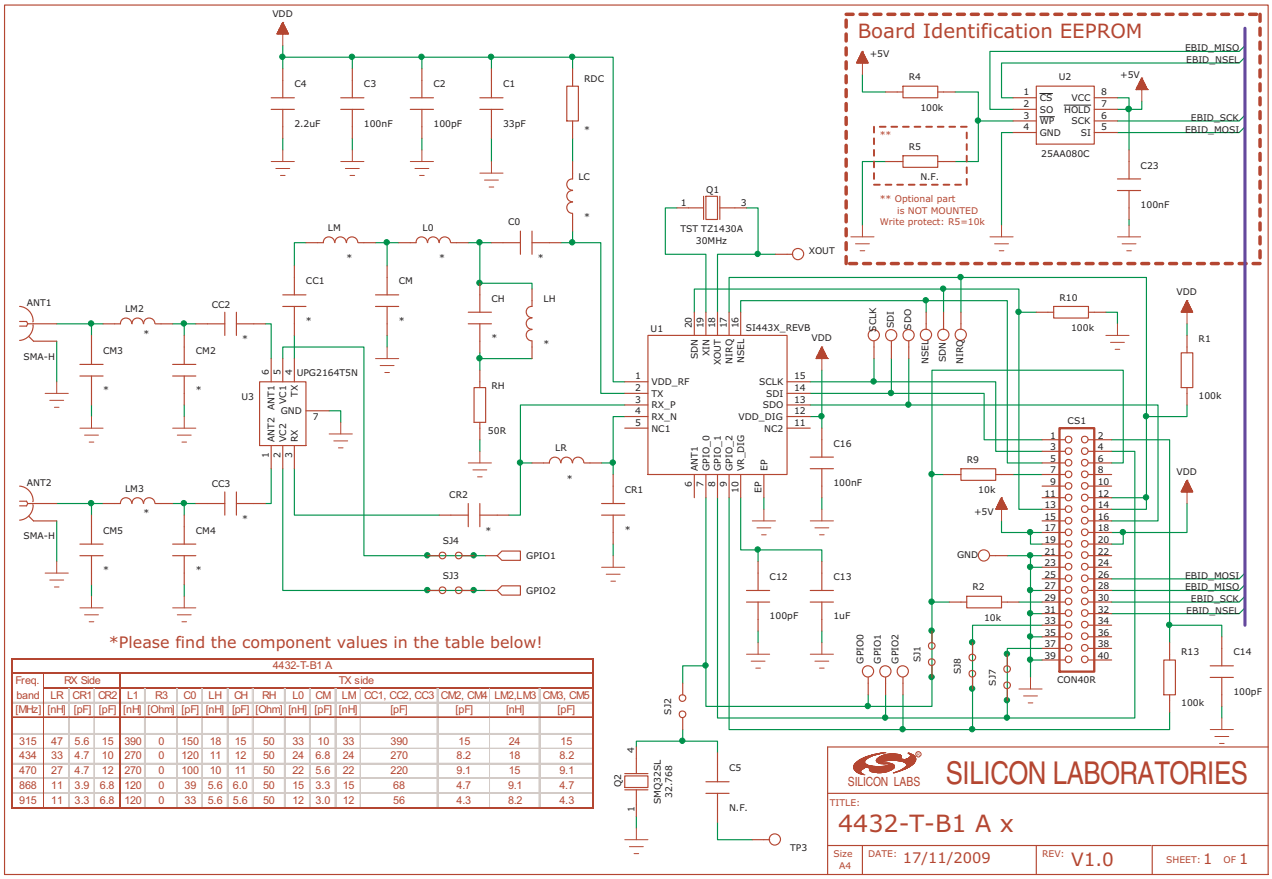


Figure 2. Direct Tie Si4431 Testcard Schematic

One of the key advantages of the EZRadioPRO devices is the built-in antenna diversity support in which two different polarization antennas are connected to the radio. At the beginning of the packet reception, the chip evaluates the received signal strength level for both antennas and uses the better one to receive the remainder of the data packet. By selecting the strongest antenna, receiver performance in the presence of multipath fading and changing antenna polarization can be greatly improved.

When using this feature, an RF switch is needed to connect the antennas to the receive or transmit path. EZRadioPRO devices control the RF switch automatically through any two GPIOs. By routing the Antenna1 Switch used for antenna diversity and the Antenna2 Switch used for antenna diversity signals to the GPIOs, the radio automatically switches between antennas during receive mode. The antenna used for reception of the RX packet is used for the subsequent TX packet transmission. On the Antenna Diversity Testcard, GPIO1 is routed to the Antenna1 Switch used for antenna diversity signals, and GPIO2 is routed to the Antenna2 Switch used for antenna diversity signals for controlling the RF switch.



4432-T-B1 A																
Freq. band	RX Side			TX side												
	LR	CR1	CR2	L1	R3	C0	LH	CH	RH	L0	CM	LM	CC1, CC2, CC3	CM2, CM4	LM2,LM3	CM3, CM5
[MHz]	[nH]	[pF]	[pF]	[nH]	[Ohm]	[pF]	[nH]	[pF]	[Ohm]	[nH]	[pF]	[nH]	[pF]	[pF]	[nH]	[pF]
315	47	5.6	15	390	0	150	18	15	50	33	10	33	390	15	24	15
434	33	4.7	10	270	0	120	11	12	50	24	6.8	24	270	8.2	18	8.2
470	27	4.7	12	270	0	100	10	11	50	22	5.6	22	220	9.1	15	9.1
868	11	3.9	6.8	120	0	39	5.6	6.0	50	15	3.3	15	68	4.7	9.1	4.7
915	11	3.3	6.8	120	0	33	5.6	5.6	50	12	3.0	12	56	4.3	8.2	4.3

Figure 3. Antenna Diversity Si4432 Testcard Schematic

The example source code for the Software Development board can be compiled for various Testcards. The following compiling options, located at the beginning of the *main_sdbc_dk3.c* file, select the proper configuration for the different testcards:

- TX/RX Split or Direct Tie Antenna configuration `#define SEPARATE_RX_TX`
- Single Antenna with RF switch `#define ONE_SMA_WITH_RF_SWITCH`
- Antenna Diversity `#define ANTENNA_DIVERSITY`

There is no compiling option in the source code of the EZLink platform since it is a fixed hardware configuration. From a radio functionality point of view, it is equivalent to a Single Antenna with RF Switch testcard.

Note: The transmit output of the EZRadioPRO devices has to be terminated properly before output power is enabled. This is accomplished by using a proper antenna or connecting the power amplifier to an RF instrument that provides 50 Ω termination to ensure proper operation and protect the device from damage.

2.2. Initializing the MCU

The software examples use a minimal set of the microcontroller's available hardware peripherals to minimize the complexity of the source code. The following hardware support is provided:

- Packet transmission is initiated by pressing a button.
- Packet transmission and packet reception are indicated via LED.
- The EZRadioPRO devices are connected via SPI to the MCU with the nIRQ pin of the radio connected to an external interrupt pin of the MCU.

While the SPI and nIRQ connections between the EZRadioPRO device and the MCU are the same on both hardware platforms, there are some differences in the peripherals:

- The example source code uses only the push button(s) and LED(s) on both platforms; however, the Software Development board has four LEDs and four push buttons, while the EZLink has only one of each. The LEDs and push buttons are on different IO pins on the two platforms.
- The PWRDN pin of the radio is connected to the MCU on the EZLink platform; so, the MCU can use it to control the radio. This connection is not available on the Software Development board, and the PWRDN pin is connected to the GND.

2.2.1. Initialization of the Software Development Board

The MCU-dependent header files are included in the source code to provide the macro definitions allowing the code to be compiled with several common C compilers (like Keil, SDCC, IAR, etc.).

```
/* ============================================================ *
 *                          INCLUDE                             *
 * ============================================================ */
#include "C8051F930_defs.h"
#include "compiler_defs.h"

/* ============================================================ *
 * C8051F930 Pin definitions for Software Development Board     *
 *                (using compiler_def.h macros)                 *
 * ============================================================ */
```

The following macros assign a label to the given IO port; so, they can be more easily referenced in the source code. For example, the first line assigns the "NSS" label to Port1.3 where the nSEL pin of the radio is connected:

```
SBIT(NSS, SFR_P1, 3);
SBIT(NIRQ, SFR_P0, 6);
SBIT(PB1, SFR_P0, 0);
SBIT(PB2, SFR_P0, 1);
SBIT(PB3, SFR_P2, 0);
SBIT(PB4, SFR_P2, 1);
SBIT(LED1, SFR_P1, 4);
SBIT(LED2, SFR_P1, 5);
SBIT(LED3, SFR_P1, 6);
SBIT(LED4, SFR_P1, 7);
```

The following definitions are used to configure the type of testcard plugged into the Software Development board.

```
//One out of these definitions has to be uncommented which tells to the compiler what kind
//of Testcard is plugged into the Software Development board
#define SEPARATE_RX_TX
//#define ANTENNA_DIVERSITY
//#define ONE_SMA_WITH_RF_SWITCH
```

Following are the function prototypes for MCU initialization and SPI functions.

```
/* ======================================================== *
 *                  Function PROTOTYPES       *
 * ======================================================== */
//MCU initialization
void MCU_Init(void);
//SPI functions
void SpiWriteRegister (U8, U8);
U8 SpiReadRegister (U8);
```

The ***void MCU_Init(void)*** function initializes all the necessary peripherals for the Software Development board:

```
void MCU_Init(void)
{
    //Disable the Watch Dog timer of the MCU
    PCA0MD   &= ~0x40;

    // Set the clock source of the MCU: 10MHz, using the internal RC osc.
    CLKSEL    = 0x14;

    // Initialize the IO ports and the cross bar
    P0SKIP  |= 0xCF;                // skip P0.0-3 & 0.6-7

    XBR1    |= 0x40;                // Enable SPI1 (3 wire mode)
    P1MDOUT |= 0x01;                    // Enable SCK push pull
    P1MDOUT |= 0x04;                    // Enable MOSI push pull
    P1SKIP  |= 0x08;                // skip NSS
    P1MDOUT |= 0x08;                    // Enable NSS push pull
    P1SKIP  |= 0xF0;                // skip LEDs
    P1MDOUT |= 0xF0;                    // Enable LEDS push pull
    P2SKIP  |= 0x03;                // skip PB3 & 4
    SFRPAGE = CONFIG_PAGE;
    P1DRV   = 0xFD;             // MOSI, SCK, NSS, LEDs high current mode
    SFRPAGE = LEGACY_PAGE;
    XBR2    |= 0x40;                // enable Crossbar

    // For the SPI communication the hardware peripheral of the MCU is used
    //in 3 wires Single Master Mode. The select pin of the radio is controlled
    //from software
    SPI1CFG   = 0x40;                           //Master SPI, CKPHA=0, CKPOL=0
    SPI1CN    = 0x00;                           //3-wire Single Master, SPI enabled
    SPI1CKR   = 0x00;
    SPI1EN  = 1;                // Enable SPI interrupt
    NSS = 1;

    // Turn off the LEDs
    LED1 = 0;
    LED2 = 0;
    LED3 = 0;
    LED4 = 0;
}
```

The ***void SpiWriteRegister(U8 reg, U8 value)*** function writes a new value into a register on the radio. If only one register of the radio is written, a 16-bit value must be sent to the radio (the 8-bit address of the register followed by the new 8-bit value of the register). To write a register, the MSB of the address is set to 1 to indicate a device write.

```
void SpiWriteRegister (U8 reg, U8 value)
{
        // Send SPI data using double buffered write

        //Select the radio by pulling the nSEL pin to low
        NSS = 0;

        //write the address of the register into the SPI buffer of the MCU
        //(important to set the MSB bit)
        SPI1DAT = (reg|0x80);           //write data into the SPI register
        //wait until the MCU finishes sending the byte
        while( SPIF1 == 0);
        SPIF1 = 0;
        //write the new value of the radio register into the SPI buffer of the MCU
        SPI1DAT = value;                        //write data into the SPI register
        //wait until the MCU finishes sending the byte
        while( SPIF1 == 0);                     //wait for sending the data
        SPIF1 = 0;
         //Deselect the radio by pulling high the nSEL pin
        NSS = 1;
}
```

The ***U8 SpiReadRegister(U8 reg)*** function reads one register from the radio. When reading a single register of the radio, a 16 bit value must be sent to the radio (the 8-bit address of the register followed by a dummy 8-bit value). The radio provides the value of the register during the second byte of the SPI transaction. Note that it is crucial to clear the MSB of the register address to indicate a read cycle.

```
U8 SpiReadRegister (U8 reg)
{

        //Select the radio by pulling the nSEL pin to low
        NSS = 0;
        //Write the address of the register into the SPI buffer of the MCU
        //(important to clear the MSB bit)
        SPI1DAT = reg;                          //write data into the SPI register
        //Wait until the MCU finishes sending the byte
        while( SPIF1 == 0);
        SPIF1 = 0;
        //Write a dummy data byte into the SPI buffer of the MCU. During sending
        //this byte the MCU will read the value of the radio register and save it
        //in its SPI buffer.
        SPI1DAT = 0xFF;                         //write dummy data into the SPI register
        //Wait until the MCU finishes sending the byte
        while( SPIF1 == 0);
        SPIF1 = 0;
        //Deselect the radio by pulling high the nSEL pin
        NSS = 1;

        //Read the received radio register value and return with it
        return SPI1DAT;
}
```

2.2.2. Initialization of the EZLink Platform

The initialization of the EZLink platform is very similar to the Software Development board and differs only in the control of the EZRadioPRO PWRDN pin. By driving this pin high, the radio is completely switched off allowing for minimal power consumption, but the values of the registers are not kept. Driving this pin low enables the radio and performs a power on reset cycle, which takes about 15 ms. The radio is not ready to receive any SPI commands in power down mode or during the power on reset cycle. That is the reason for a delay in the main function after the MCU is initialized and the PWRDN pin of the radio is driven low.

```
//Initialize the MCU:
//          - set IO ports for the Software Development board
//          - set MCU clock source
//          - initialize the SPI port
//          - turn off LEDs
MCU_Init();
          /* ============================================================ *
           *                    Initialize the Si4432 ISM chip     *
           * ============================================================ */
//Turn on the radio by pulling down the PWRDN pin
SDN = 0;
//Wait at least 15ms before any initialization SPI commands are sent to the radio
// (wait for the power on reset sequence)
for (temp8=0;temp8<15;temp8++)
{
          for(delay=0;delay<10000;delay++);
}
//read interrupt status registers to clear the interrupt flags and release NIRQ pin
ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register
```

3. Packet Transmission Using the Transmit Packet Handler

This software example uses push buttons on the hardware platform to initiate a packet transmission. After power on reset, the firmware initializes the MCU and the EZRadioPRO chip. The chip is set to IDLE mode, where only the crystal oscillator of the radio is running, and the software waits for a button press. If any of the push buttons are pressed on the Software Development board, the demo sends a packet then returns to polling the push buttons. The packet configuration used for the transmission is as shown in Table 1.

Table 1. Demo Code Packet Configuration

Preamble					Synchron pattern		Payload length	Payload: “Button1+CR”								CRC	
55	55	55	55	55	2D	D4	08	42	75	74	74	6F	6E	31	0D	7A	B1
5 bytes (10 bytes if Ant. Diversity is used)					2 bytes		1 byte	8 bytes								2 bytes	

The payload of the packet is set according to the push button pressed.

Note: Since there is only one push button on the EZLink platform, it always sends the “Button1” payload.

While the only real application information in the packet is the string that tells which button is pressed, the actual packet requires several additional bytes around the payload for proper operation. These additional packet fields include a preamble, sync word, payload length, and CRC.

After enabling the receiver on the radio, it will start to receive data even if there is no signal present. In order for the receiver to detect that a valid packet is present, the transmitted packet should start with a preamble and a synchron pattern.

- The *preamble* is a continuous 0101 sequence used to synchronize the transmitter and receiver. The preamble is a known sequence with continuous edge changes in the data so that the demodulator and clock recovery circuit of the receiver node can be settled correctly. The minimum required preamble length is application-dependent. See "4. Packet Receiving Using the Receive Packet Handler" on page 18 for more details.
- After the transmitter and receiver have synchronized, the receiver has to find out where the payload data in the packet starts. The transmitter includes a known bit pattern to help identify the payload data, the *synchron word*.

The receiver also needs to know how long the transmitted packet is. There are several possibilities that can be used to indicate the length of the packet:

- Send a special end character at the end of the packet.
- Always transmit a fixed length packet.
- Include the length information in the transmitted packet.

In the example code, the length of the payload is included in the packet.

In the RF physical channel, the transmitted packet can become corrupted by noise or another RF transmitter can send a packet at the same instant; so, it is strongly recommended that some kind of error checking be used to confirm whether the received data is valid or contains errors.

If the communication is disturbed, it can cause a burst of bit errors, which can be identified by using a CRC. While a simple checksum can be used to recognize single-bit errors, most link disruptions result in burst errors, which require the CRC for robust detection.

3.1. Initialization of the Radio

The EZRadioPRO radio has a built-in packet handler for automatic packet transmission and reception. Once the basic packet parameters are initialized, only the payload data needs to be sent to the onboard FIFO of the radio, and packet transmission will automatically start. The entire packet is built up and sent by the radio without the use of the microcontroller. When the packet transmission is finished, the radio generates an interrupt for the microcontroller and goes back to the previously-active state.

This section describes how the EZRadioPRO transmitter or transceiver devices can be configured to send the packet shown in Table 1. The *tx_SDBC_DK3* or *tx_EZlink* example projects are used to demonstrate packet transmission. The code segments used in this chapter are copied from the *main_sdbc_dk3.c* or *main_ezlink.c* files.

The code section for initialization the radio must be performed in case of the following:

- A reset event occurs.
- The power-down pin of the radio is pulled low (radio enabled).

Note: The power-on reset cycle of the EZRadioPRO devices takes about 15 ms. During this reset period, the radios cannot accept any SPI command. There are two ways to determine if the chip is ready to receive SPI commands after a reset event:

- Use a timer in the microcontroller to wait for 15 ms.
- Enable the external interrupt of the microcontroller and wait for an interrupt from the radio. In the MCU ISR routine, check the “ipor” interrupt status bit, which will be set if the POR event finished correctly.

3.1.1. Software Reset for the Radio

If the microcontroller handles the power-on reset for the entire application, it is recommended that a software reset be provided for the radio every time the microcontroller performs a power-on reset sequence.

The following code section shows how to perform a software reset for the radio and determine when the reset procedure is finished, allowing the radio to receive SPI commands from the microcontroller:

```
//SW reset
SpiWriteRegister(0x07, 0x80);               //write 0x80 to the Operating & Function Control1 register
//wait for chip ready interrupt from the radio (while the nIRQ pin is high)
while ( NIRQ == 1);
//read interrupt status registers to clear the interrupt flags and release NIRQ pin
ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register
```

3.1.2. Set RF Parameters

The following section sets the basic RF parameters needed to do OOK, FSK, or GFSK modulated packet transmission. These parameters include center frequency, transmit data rate, and transmit deviation.

```
        /*set the physical parameters*/
//set the center frequency to 915 MHz
SpiWriteRegister(0x75, 0x75);     //write 0x75 to the Frequency Band Select register
SpiWriteRegister(0x76, 0xBB);     //write 0xBB to the Nominal Carrier Frequency1 register
SpiWriteRegister(0x77, 0x80);     //write 0x80 to the Nominal Carrier Frequency0 register

//set the desired TX data rate (9.6kbps)
SpiWriteRegister(0x6E, 0x4E);     //write 0x4E to the TXDataRate 1 register
SpiWriteRegister(0x6F, 0xA5);     //write 0xA5 to the TXDataRate 0 register
SpiWriteRegister(0x70, 0x2C);     //write 0x2C to the Modulation Mode Control 1 register

//set the desired TX deviation (+-45 kHz)
```

```
SpiWriteRegister(0x72, 0x48);        //write 0x48 to the Frequency Deviation register
```

Note: For OOK modulation, there is no need to configure the transmit deviation.

Besides the TX Data Rate 0 and TX Data Rate 1 registers, an additional bit is used to define the data rate: “txdtrtscale” (bit5 in the Modulation Mode Control1 register). The “txdtrtscale” bit must be set to 1 if the desired data rate is below 30 kbps.

The MSB of the deviation setting can be found in the Modulation Mode Control register 2 (fd[8] – bit2).

3.1.3. Set Packet Configuration

The EZRadioPRO devices provide a flexible packet handler that can support a wide range of packet configurations including the following:

- Programmable preamble (up to 255 bytes)
- Programmable synchronization word length and pattern (up to four bytes)
- Automatic header generation and qualification (up to four bytes)
- Fixed and variable length packets
- Payload data of up to 64 bytes (using the built-in FIFO)
- Automatic CRC calculation and verification (with support for three different CRC polynomials)

If the user’s application operates with a packet configuration conforming to the above options, the EZRadioPRO packet handler can be used. Using the packet handler allows the MCU to configure the transmit packet format and structure during configuration. When a packet is transmitted, the MCU only needs to place the payload data into the TX FIFO before each packet transmission and the packet handler will automatically construct and transmit the packet.

Note: If the user's application operates with a packet configuration that is not supported by the packet handler, the EZRadio-PRO devices can be operated in the following modes as well:
- Direct mode or Synchronous Direct mode (check the EZRadioPRO data sheet for more details).
- RAW data mode (See “AN463: RAW Data Mode with EZRadioPRO®” for more details).

Another powerful feature of the EZRadioPRO devices is Automatic Antenna Diversity. Antenna diversity allows the receiver to use two separate antennas to help combat multi-path fading or antenna polarization effects. The receiver will automatically evaluate the signal strength on both antennas and select the antenna with maximum power. While antenna diversity is used during receiving mode, the transmitter needs to be properly configured for the system to support diversity. The following considerations need to be followed for antenna diversity operation:

- A longer preamble is required to allow the receiver to evaluate the signal strength level on both antennas to determine which is optimal for receiving the packet.
- The built-in antenna diversity algorithm uses the GPIO pins to control the RF switch (for selecting the proper antenna). Two GPIOs must be configured to route the control signals for the RF switch.
- The proper signal polarity of the RF switch control signal must be configured.

The example code shown below demonstrates how to configure the packet handler for the sample packet shown in Table 1.

The example can be compiled for Antenna Diversity or non-Antenna diversity operation. The selection is made via the following configuration options as described in "2. Hardware Options" on page 1: SEPARATE_RX_TX, ANTENNA_DIVERSITY, and ONE_SMA_WITH_RF_SWITCH.

The preamble length is set depending on whether the antenna diversity algorithm is enabled:

- When antenna diversity is disabled, a 5 byte preamble is transmitted. The shorter preamble allows the receiver to enable auto-frequency calibration (AFC) with a preamble detection threshold of two bytes. (Receiver configuration is described in more detail in "4. Packet Receiving Using the Receive Packet Handler" on page 18.)
- When antenna diversity is enabled, a longer preamble must be transmitted. The longer preamble allows additional time for the receiver to evaluate the signal strength for both antennas, settle the clock recovery circuit correctly, etc.

```
                    /*set the packet structure and the modulation type*/
        //set the preamble length to 10bytes if the antenna diversity is used and set to 5bytes if not
#ifdef ANTENNA_DIVERSITY
        SpiWriteRegister(0x34, 0x14);       //write 0x14 to the Preamble Length register
#else
        SpiWriteRegister(0x34, 0x09);       //write 0x09 to the Preamble Length register
#endif
```

Disable the header bytes (not used in this example) and set the synchron word length to two bytes.

Note: The MSB bit of the preamble length setting is located in this register (“prealen8”—bit0 in the Header Control2 register).

```
        //Disable header bytes; set variable packet length (the length of the payload is defined by the
        //received packet length field of the packet); set the synch word to two bytes long
        SpiWriteRegister(0x33, 0x02);       //write 0x02 to the Header Control2 register
```

Set the synchron word pattern for 0x2DD4. At least two bytes of synchron word are recommended to increase the robustness of the communication. The synchron word pattern is a system-level design consideration. For interoperability with EZRadio devices, the synchron word must be set to 0x2DD4. The synchron word can also be used as a packet filter by using different values for different applications or node types. Different synchron word vales can be used by the receiver to filter and receive only communication from the desired node.

```
        //Set the sync word pattern to 0x2DD4
        SpiWriteRegister(0x36, 0x2D);       //write 0x2D to the Sync Word 3 register
        SpiWriteRegister(0x37, 0xD4);       //write 0xD4 to the Sync Word 2 register
```

Enable the TX packet handler and the CRC16 calculation:

```
        //enable the TX packet handler and CRC-16 (IBM) check
        SpiWriteRegister(0x30, 0x0D);       //write 0x0D to the Data Access Control register
```

This example will use the EZRadioPRO 64-byte TX FIFO to provide the data for the TX packets. By using the FIFO, the MCU simply provides the data to fill the FIFO; the radio generates the correct bit timing internally and modulates the output signal. The FIFO can be used with the TX packet handler to construct the packet format, or the entire packet can be calculated in the MCU and placed into the TX FIFO.

When the TX packet handler is not used, no packet forming is provided by the radio. The MCU should construct the entire packet (including preamble, synchron word, header fields, payload, etc.) and fill the formed packet into the transmit FIFO.

EZRadioPRO also supports several modes to provide the packet data as data bits to the radio:

- **Using one of the GPIOs for direct modulation**
  In this case the MCU has to form the packet and provide the data bits with the right bit timing to the selected GPIO.
- **Using the SDI pin for direct modulation**
  In this case the MCU has to form the packet and provide the data bits with the right bit timing to the SDI pin.
- **Using an internal PN9 random data generator**
  This is a test mode that can be used to measure the shape of the modulated output spectrum.

In the current example, the FIFO is the source of the modulation and the TX packet handler is used to form and transmit the packet, the Modulation Mode Control2 register is set accordingly:

```
//enable FIFO mode and GFSK modulation
SpiWriteRegister(0x71, 0x63);//write 0x63 to the Modulation Mode Control 2 register
```

If antenna diversity is used, the correct GPIO configuration must be selected. For the Si4432-T-B1-A-xxx Testcard, use the following configuration for the RF switch:

```
#ifdef ANTENNA_DIVERSITY
    SpiWriteRegister(0x0C, 0x17);     //write 0x17 to the GPIO1 Configuration(set the Antenna 1 Switch used for antenna diversity )
    SpiWriteRegister(0x0D, 0x18);     //write 0x18 to the GPIO2 Configuration(set the Antenna 2 Switch used for antenna diversity )
#endif
#ifdef ANTENNA_DIVERSITY
    //enable the antenna diversity mode
    SpiWriteRegister(0x08, 0x80);     //write 0x80 to the Operating Function Control 2 register
#endif
```

For the Single Antenna with RF Switch Testcard, use the following configuration for GPIO1 and GPIO2 to control the RF switch:

```
#ifdef ONE_SMA_WITH_RF_SWITCH
     SpiWriteRegister(0x0C, 0x12);//write 0x12 to the GPIO1 Configuration(set the TX state)
    SpiWriteRegister(0x0D, 0x15);//write 0x15 to the GPIO2 Configuration(set the RX state)
#endif
```

3.1.4. Select Modulation

EZRadioPRO supports three different modulation types:

- Frequency shift keying (FSK)
- Gaussian frequency shift keying (GFSK)
- On-off keying (OOK)

This example uses GFSK modulation, but the following sections provide an overview of the different modulation types.

Note: The EZRadioPRO devices can be set to provide an unmodulated carrier signal for test purposes by setting the “modtyp[1:0]” bits to 0 (bit[1:0] in the Modulation mode Control2 register).

3.1.4.1. Frequency Shift Keying

FSK modulation uses a change in the frequency of the signal to transmit digital data.

- Without modulation, the radio transmits a continuous wave signal (called the CW signal) on the center frequency.
- To send a 0 bit, the CW signal is decreased in frequency by an amount equal to the deviation, resulting in a frequency of $f_0 - \Delta f_{FSK}$, where f_0 is the center frequency and Δf_{FSK} is the deviation.
- To send a 1 bit, the CW signal is increased in frequency by an amount equal to the deviation, resulting in a frequency of $f_0 + \Delta f_{FSK}$.

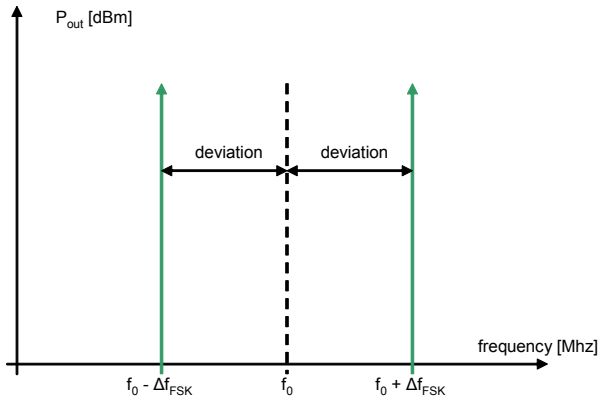


Figure 4. FSK Modulation

To enable FSK modulation, the “modtyp[1:0]” bits in the Modulation Mode Control2 register are set to 0x2.

FSK modulation is robust against interferers; however, the performance depends on the accuracy of the frequency reference and thus on the crystal used with the radio.

- A crystal of 10–20 ppm accuracy is recommended.
- Crystals with looser tolerances can be used but require an increase in TX deviation and receiver bandwidth to ensure that signal energy falls within the receiver’s filter bandwidth. Increasing the signal bandwidth results in a *decrease* in system sensitivity.

3.1.4.2. Gaussian Frequency Shift Keying

GFSK modulation works like FSK modulation except that the data bits are filtered with a Gaussian filter. This filtering reduces the sharp edges of the TX bits resulting in reduced spectral spreading and a narrower occupied bandwidth.

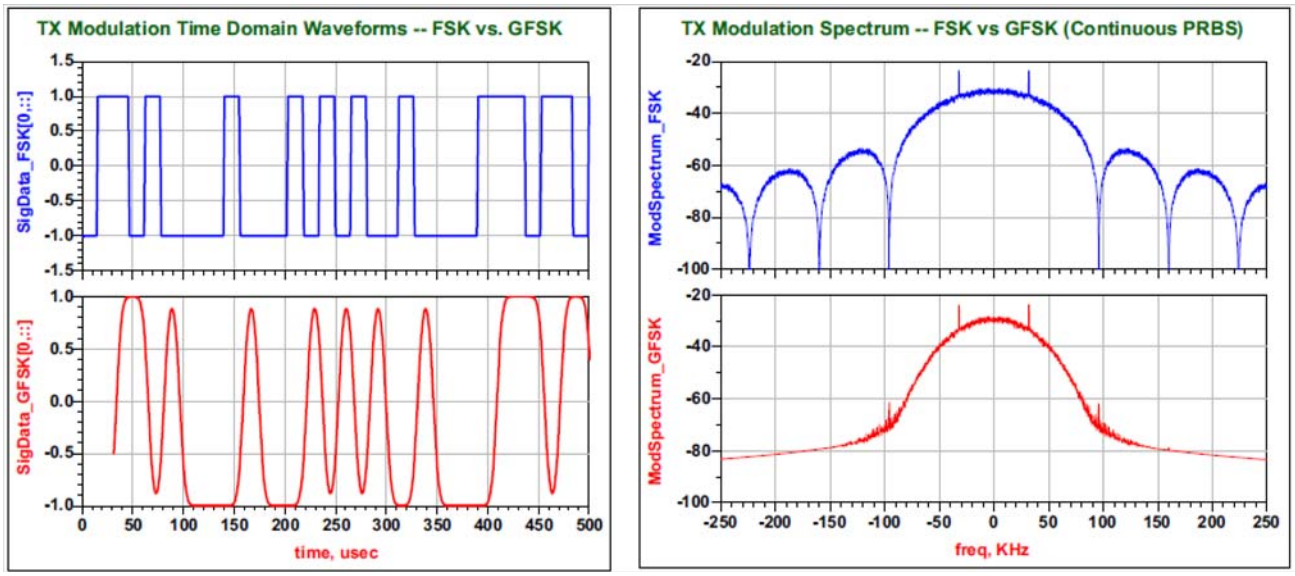


Figure 5. FSK and GFSK Modulation Differences (Time Domain and Spectrum)

GFSK modulation is enabled by setting the “modtyp[1:0]” bits in the Modulation Mode Control2 register to 0x3.

GFSK modulation offers robust link performance providing the highest performance in the narrowest occupied bandwidth. Like FSK, the radio performance and link parameters are dependent on the selected crystal or TCXO.

3.1.4.3. On-Off Keying

OOK modulation encodes the data by switching on and off the power amplifier:

- When no data is present, the power amplifier is switched off.
- To send a 0 bit, the power amplifier is switched off during the bit period.
- To send a 1 bit, the power amplifier is switched on during the bit period.

OOK modulation is enabled by setting the"modtyp[1:0]" bits in the Modulation Mode Control2 register to 0x1.

OOK modulation consumes less current compared to FSK and GFSK modulation since the power amplifier is switched off when transmitting a 0. OOK modulation provides less link robustness than FSK or GFSK modulation and requires more frequency bandwidth. As a result, OOK modulation is normally only used if compatibility with an existing product is required.

3.1.5. Registers Configuration for Si4432 Revision V2

The Si4432 Revision V2 requires some registers to be programmed to values other than their default values:

The PLL and VCO must be programmed to the following settings:

```
        /*set the non-default Si4432 registers*/
//set VCO and PLL
SpiWriteRegister(0x5A, 0x7F);        //write 0x7F to the VCO Current Trimming register
SpiWriteRegister(0x59, 0x40);        //write 0x40 to the Divider Current Trimming register
```

Note: These settings are not required for subsequent versions of the radio

3.1.6. Registers Configuration for Si4431 Revision A0

The Si4431 revision A0 requires some registers to be programmed to values other than their default value. The PLL and VCO must be programmed to the following values to get optimal current consumption:

```
//set VCO and PLL
SpiWriteRegister(0x57, 0x01);//write 0x01 to the Chargepump Test register
SpiWriteRegister(0x59, 0x00);//write 0x00 to the Divider Current Trimming register
SpiWriteRegister(0x5A, 0x01); //write 0x01 to the VCO Current Trimming register
```

Note: These settings are not required for subsequent versions of the radio.

3.1.7. Crystal Oscillator Tuning Capacitor

The accuracy of the radio center frequency is determined by several parameters of the crystal used with the radio, such as load capacitance, crystal accuracy, and the parasitic of the PCB associated with the crystal circuit.

EZRadioPRO provides several features to reduce the impact of these crystal parameters:

- By using a wide transmit deviation and wide receiver bandwidth, the link will be less sensitive to any frequency offset, but the link budget will be reduced compared to an ideal case.
- By using the built-in Auto-frequency calibration, the radio will adjust its center frequency to align with the transmitter center frequency. This approach has the limitation of requiring a longer preamble.
- Crystal inaccuracy can be compensated for by tuning the Crystal Oscillator Load Capacitance register to the proper capacitance.

  The Crystal Oscillator Load Capacitance register can be used to compensate for crystal mismatch by tuning it to the proper capacitance value.

  Crystal inaccuracy can be compensated for by tuning the Crystal Oscillator Load Capacitance register to the proper capacitance.

  Assuming that the same PCB and type of crystal is used during the entire life cycle of the product, the crystal oscillator load capacitance value can be measured once and programmed.

On the Silicon Labs demo boards, the same type of crystal is used. The proper crystal load capacitance was measured:

```
//set Crystal Oscillator Load Capacitance register
SpiWriteRegister(0x09, 0xD7);       //write 0xD7 to the CrystalOscillatorLoadCapacitance register
```

3.1.7.1. Tips

Two simple methods can be used to determine the correct crystal load capacitance value:

- Measure the center frequency of the radio with a spectrum analyzer and adjust the Crystal Oscillator Load Capacitance register until the center frequency is aligned to the desired frequency.
- Select the microcontroller clock output to GPIO2 and measure the clock using a frequency counter. Adjust the Crystal Oscillator Load Capacitance register until the measured clock is at the desired frequency.

3.2. Send a Packet

After the MCU and the radio are initialized for packet transmission the software enters into a continuous loop looking for a button press. Once a button is pressed, the software fills the appropriate payload into the TX FIFO, starts the packet transmission, and waits for the “enpksent” interrupt. During packet transmission, one of the LEDs is turned on. The following code section shows how a button press is captured and how the packet is sent (the same code segment is used for the other buttons as well).

The main loop of the source code is as follows:

```
/*MAIN Loop*/
while(1)
{
        //Poll the port pins of the MCU to figure out whether the push button is pressed or not
        if(PB1 == 0)
        {
                //Wait for releasing the push button
                while(PB1 == 0);
                //turn on the LED to show the packet transmission
                LED1 = 1;
                /*SET THE CONTENT OF THE PACKET*/
                //set the length of the payload to 8bytes
                SpiWriteRegister(0x3E, 8);          //write 8 to the Transmit Packet Length register
                //fill the payload into the transmit FIFO
                SpiWriteRegister(0x7F, 0x42);       //write 0x42 ('B') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x55);       //write 0x55 ('U') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x54);       //write 0x54 ('T') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x54);       //write 0x54 ('T') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x4F);       //write 0x4F ('O') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x4E);       //write 0x4E ('N') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x31);       //write 0x31 ('1') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x0D);       //write 0x0D (CR) to the FIFO Access register

                //Disable all other interrupts and enable the packet sent interrupt only.
                //This will be used for indicating the successful packet transmission for the MCU
                SpiWriteRegister(0x05, 0x04);       //write 0x04 to the Interrupt Enable 1 register
                SpiWriteRegister(0x06, 0x00);       //write 0x00 to the Interrupt Enable 2 register
                //Read interrupt status registers. It clear all pending interrupts and the nIRQ pin goes back to high.
                ItStatus1 = SpiReadRegister(0x03);  //read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);  //read the Interrupt Status2 register

                /*enable transmitter*/
                //The radio forms the packet and send it automatically.
                SpiWriteRegister(0x07, 0x09);       //write 0x09 to the Operating Function Control 1 register

                /*wait for the packet sent interrupt*/
                //The MCU just needs to wait for the 'ipksent' interrupt.
                while(NIRQ == 1);
                //read interrupt status registers to release the interrupt flags
                ItStatus1 = SpiReadRegister(0x03);  //read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);  //read the Interrupt Status2 register

                //wait a bit for showing the LED a bit longer
                for(delay = 0; delay < 10000;delay++);
                //turn off the LED
                LED1 = 0;
        }
```

4. Packet Receiving Using the Receive Packet Handler

The software example works as a continuous receiver node. After the MCU and the EZRadioPRO receiver or transceiver device is configured, the firmware sets the radio into continuous receiving mode and waits for packets. Once a packet arrives with the same packet configuration shown in Table 1 on page 9, it blinks the appropriate LED. If a packet is received with a payload “Button1”, the software blinks LED1 on the Software Development board (LED2 for “Button2”, etc.). After a successful packet reception, the software restarts the receiver and waits for the next packet.

The software example can be used as a receiver node for the software example in "3. Packet Transmission Using the Transmit Packet Handler" on page 9.

Note: The EZLink platform accepts packets with a “Button1” payload only.

4.1. Initialization of the Radio

The EZRadioPRO receiver and transceiver devices have a built-in digital modem that can be configured for different radio settings. It supports OOK, FSK, and GFSK modulations as well as variable baseband filter bandwidths for narrow or wide band applications. The digital modem demodulates the received data bits and provides them to the application. There are several methods that can be used to obtain the received bits from the chip:

- **Direct Mode**
  The EZRadioPRO devices can provide the raw data bits on one of the GPIOs. In this case, the microcontroller must handle the received data bit-by-bit as it is sent from the radio. This processing requires the MCU to consume cycles packaging, decoding, and qualifying the data. In synchronous direct data mode, the radio provides the bit clock for the raw data to ease sampling by the MCU. In asynchronous direct mode, the bit clock is ignored.
- **FIFO Mode**
  There is a 64 byte receive FIFO available in the EZRadioPRO devices. After a successful preamble and synchron word recognition, the chip fills the received data bits into the RX FIFO directly. The microcontroller can access the FIFO by using standard SPI commands (reading the FIFO Access register). There are additional built-in functions available for easier FIFO handling. The RX FIFO almost full status signal can alert the microcontroller when to read data from the FIFO. Its threshold is programmable through the RF FIFO Almost Full Threshold register in 1…64 bytes.
- **FIFO Mode with RX Packet Handler**
  If the format of the packet used in the application fits into the options listed in "3.1.3. Set Packet Configuration " on page 11, the radio can be set to receive the packet automatically. This is the best option since the resources of the MCU are not used during the packet reception. The radio provides an interrupt to the MCU if the packet is received correctly, allowing the MCU to sleep or perform other functions until data are available from the radio.

In the current software example, the RX Packet Handler is used for receiving the packets. The following section shows how the radio is configured for this mode.

4.1.1. Preamble Detection

It is mandatory to start every packet with a preamble (“0101...” bit pattern). The role of the preamble is to allow the receiver to properly lock to the received signal prior to the arrival of the data payload. The minimum required length of the preamble is determined from the sum of the receiver settling time and the preamble detection threshold.

The receiver settling when using FSK or GFSK modulation depends upon whether the auto-frequency calibration (AFC) is enabled. The minimum receiver settling time is 1 byte, but, if the AFC is enabled, the receiver settling time is 2 bytes. Also, in case of OOK modulation, the receiver settling time is 2 bytes.

The EZRadioPRO receivers have a built-in Preamble Detector. After settling the receiver, the radio compares the received bits to the preamble bit pattern. If the preamble detector finds the predefined length of consecutive preamble bits in the received data, the radio reports a valid preamble reception and can generate an “ipreavalid” interrupt for the microcontroller.

The preamble detection threshold (the length of the preamble) is programmable. The required preamble length depends on several factors including the duration of the radio on-time, antenna diversity, and AFC. If the receiver on time relative to the length of the packet is long, then a short preamble threshold may result in false preamble detection. This condition can arise since the receiver will receive long periods of random noise between packets and there is a probability that the noise, which is random data, can contain a short preamble bit pattern. In this case, a longer preamble threshold, such as 20 bits, is recommended. If the receiver is enabled synchronously just before the packet is transmitted a short preamble detection threshold, like 8 bits, can be used.

When antenna diversity is enabled, the receiver must make additional measurements during the preamble stage, and thus a longer preamble must be used. It is recommended to set the preamble detection threshold to 20 bits when using antenna diversity. Likewise when using the AFC in the receiver, additional measurements are required during the preamble stage resulting in a longer required preamble length.

Table 2 summarizes the recommended transmit preamble length and the preamble detection threshold for different cases:

Table 2. Recommended Preamble Length

Mode	Approximate Receiver Settling Time	Recommended Preamble Length with 8-Bit Detection Threshold	Recommended Preamble Length with 20-Bit Detection Threshold
(G)FSK AFC Disabled	1 byte	20 bits	32 bits
(G)FSK AFC Enabled	2 byte	28 bits	40 bits
(G)FSK AFC Disabled + Antenna Diversity Enabled	1 byte	—	64 bits
(G)FSK AFC Enabled + Antenna Diversity Enabled	2 byte	—	8 byte
OOK	2 byte	3 byte	4 byte
OOK + Antenna Diversity Enabled	8 byte	—	8 byte

4.1.1.1. Tips

If the Auto-Frequency Calibration is enabled, the Frequency Offset 1 and 2 registers hold the measured offset between the transmitter and receiver boards. The offset value can be read and stored to use to adjust out any systematic frequency offset between the receiver and transmitter. For example in applications where the length of the transmit packet is critical the length of the preamble can be shorter by using the Offset registers instead of the AFC to adjust for frequency variations between the TX and RX nodes.

The radio includes a built in Invalid Preamble detector which is used to validate the channel within the shortest time: once the receiver is enabled the invalid preamble timer is started. If preamble is not received within the invalid preamble timeout, then the radio can generate an interrupt for the microcontroller indicating that there is no packet transmission.

The timeout depends on the revision of the EZRadioPRO chip:

- If Si4431-A0 is used, then the threshold is set to

$$Timeout = Invalid\_Preamble\_Threshold[5:2] \times 4 \times T_{bit}$$

- If Si4432-V2 is used, then the threshold is set to

$$Timeout = 4 \times T_{bit} + 16 \times (Preath[4:0] + 1) \times T_{bit}$$

- If Si443x-B1 is used, then the threshold is set to

$$Timeout = (preath(4.0) + 2) \times 2 \times T_{bit}$$

This feature is useful for the following cases:

- A frequency hopping protocol implementation to validate whether there is packet transmission on the given channel or the software has to jump to the next frequency.
- For time-synchronized protocols, after waking up, the node can decide rapidly whether somebody is transmitting or it can go back to sleep mode.

4.1.2. Synchron Word Detection

After successful preamble detection, the radio waits for the synchron word. It compares the received bits to the synchron word pattern. Once the synchron word pattern matches with the received bits, the radio starts to fill the FIFO with the packet payload data. The EZRadioPRO devices can provide an “iswdet” interrupt to the MCU upon synchron word reception.

The length and the pattern of the synchron word are programmable from 1–4 bytes. It is recommended to use at least two bytes of synchron word to avoid false detection.

4.1.3. Selecting the Modulation Type

There are several registers that must be set to configure the behavior of the digital modem. These registers must be configured for the RF parameters to be used: modulation, crystal tolerance, data rate, transmit deviation, and AFC state. Knowing the relationship between these registers and the RF parameters is not required because an Excel calculator tool is provided to easily define the modem parameters based on the RF settings. The modem parameters vary between the OOK or FSK/GFSK modulation types.

4.1.3.1. Modem Configuration for OOK Modulation

For OOK modulation, the following registers must be configured:

- IF Filter Bandwidth
- Clock Recovery Oversampling Ratio
- Clock Recovery Offset 2
- Clock Recovery Offset 1
- Clock Recovery Offset 0
- Clock Recovery Timing Loop Gain 1
- Clock Recovery Timing Loop Gain 0

The input parameters of the Excel calculator are as follows:

- Modulation type is OOK
- Desired data rate
- Desired receiver baseband bandwidth
- Usage of the Manchester coding

As Figure 6 shows, the Excel calculator provides the proposed modem parameters in hexadecimal format (showing the settings in the orange cells) and it also provides the necessary WDS control commands for setting up the digital modem (“RX OOK Modem WDS COMMANDS” section).

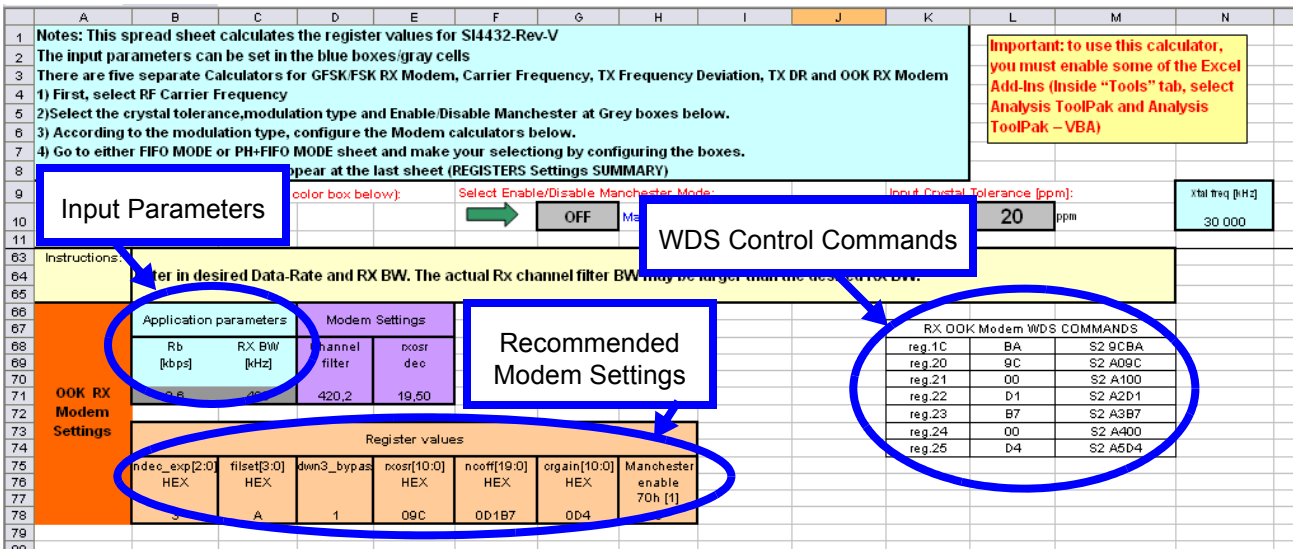


Figure 6. OOK Modem Parameter Calculation

Note: The modem parameters are different for all revisions; therefore, separate Excel calculators are provided for them.

4.1.3.2. Modem Configuration for FSK/GFSK Modulation

For FSK/GFSK modulation, the following registers must be configured:

- IF Filter Bandwidth
- Clock Recovery Oversampling Ratio
- Clock Recovery Offset 2
- Clock Recovery Offset 1
- Clock Recovery Offset 0
- Clock Recovery Timing Loop Gain 1
- Clock Recovery Timing Loop Gain 0
- AFC Loop Gearshift Override

The following are input parameters for the Excel calculator:

- Modulation type is FSK or GFSK
- Manchester coding enabled/disabled
- Crystal tolerance
- Desired data rate
- Transmit deviation

Based on the input parameters, the Excel calculator defines the proposed Channel filter bandwidth and the estimated receiver sensitivity. The Excel calculator also provides the proposed modem parameters in hexadecimal format (showing the settings in the orange cells) and the necessary WDS control commands for setting up the digital modem ("RX OOK Modem WDS COMMANDS" section).

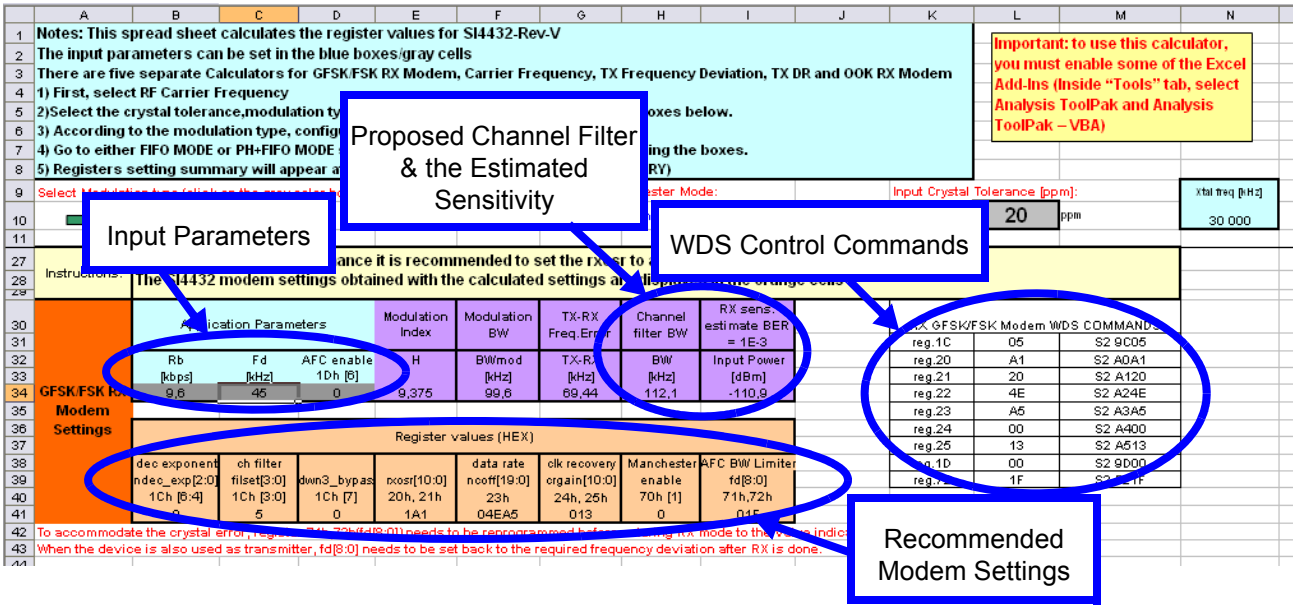


Figure 7. Modem Parameters for FSK/GFSK Modulation

Note: The modem parameters are different for all revisions; therefore, separate Excel calculators are provided for them.

4.1.4. Initializing the Radio in the Example Source Code

The following code sections are taken from the main.c files of the rx_SDBC_CK3 or rx_EZLink projects. For microprocessor initialization and Testcard selection see "2. Hardware Options" on page 1.

The example source code uses GFSK modulation, 9.6 kbps data rate, and ±45 kHz modulation, with the following code configuring the Si4432 radio accordingly.

4.1.4.1. Software Reset

Provide software reset for the radio. (The necessity of the radio reset is discussed in "3.1.1. Software Reset for the Radio" on page 10.)

```
/* ============================================================ *
 *                  Initialize the Si4432 ISM chip              *
 * ============================================================ */
//SW reset
SpiWriteRegister(0x07, 0x80);           //write 0x80 to the Operating & Function Control1 register
//wait for chip ready interrupt from the radio (while the nIRQ pin is high)
while ( NIRQ == 1);
//read interrupt status registers to clear the interrupt flags and release NIRQ pin
ItStatus1 = SpiReadRegister(0x03);      //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);      //read the Interrupt Status2 register
```

4.1.4.2. Set RF Parameters

The following code configures the physical layer parameters of the radio to work according to the desired RF settings. It configures the center frequency to 915 MHz:

```
/*set the physical signal parameters*/
//set the center frequency to 915 MHz
SpiWriteRegister(0x75, 0x75);           //write 0x75 to the Frequency Band Select register
SpiWriteRegister(0x76, 0xBB);           //write 0xBB to the Nominal Carrier Frequency1 register
SpiWriteRegister(0x77, 0x80);           //write 0x80 to the Nominal Carrier Frequency0 register
```

4.1.4.3. Tips

If the application uses multiple frequencies for the communication and these frequencies are channelized, then the EZRadioPRO devices provide a simple method to rapidly change center frequency. The frequency of the first channel is defined by the Frequency Band Select and Nominal Frequency 1 and 2 registers. The frequency step between the channels is defined by the Frequency Hopping Step Size register in 10 kHz increments. After configuring these settings, the actual channel can be set by the Frequency Hopping Channel Select register. In this case, only one SPI register needs to be written for changing the frequency.

Configure the modem parameters for receiving the desired GFSK modulated data:

```
/*set the modem parameters according to the excel calculator (parameters: 9.6 kbps, deviation: 45 kHz, channel filter BW: 112.1 kHz*/
SpiWriteRegister(0x1C, 0x05);     //write 0x05 to the IF Filter Bandwidth register
SpiWriteRegister(0x20, 0xA1);     //write 0xA1 to the Clock Recovery Oversampling Ratio register
SpiWriteRegister(0x21, 0x20);     //write 0x20 to the Clock Recovery Offset 2 register
SpiWriteRegister(0x22, 0x4E);     //write 0x4E to the Clock Recovery Offset 1 register
SpiWriteRegister(0x23, 0xA5);     //write 0xA5 to the Clock Recovery Offset 0 register
SpiWriteRegister(0x24, 0x00);     //write 0x00 to the Clock Recovery Timing Loop Gain 1 register
SpiWriteRegister(0x25, 0x13);/    /write 0x13 to the Clock Recovery Timing Loop Gain 0 register
SpiWriteRegister(0x1D, 0x40);     //write 0x40 to the AFC Loop Gearshift Override register
SpiWriteRegister(0x72, 0x1F);     //write 0x1F to the Frequency Deviation register
```

Note: The Si4432 revV2 chip uses the Frequency Deviation register to define the maximum allowable frequency offset for Auto-Frequency Calibration.
The value of this register must be set as an AFC limit before going to receive mode; Change the value of this register according to frequency deviation before going to transmit mode. The Si4431 revision A0 and Si443x revision B1 chips have a separate register for this purpose: AFC Limiter.

Note that the same RF settings (data rate, deviation, etc.) require different modem configuration for Si4431 revision A0 than Si4432 revision V2:

```
/*set the modem parameters according to the exel calculator(parameters: 9.6 kbps, deviation: 45 kHz, channel filter BW: 102.2 kHz*/
SpiWriteRegister(0x1C, 0x1E);     //write 0x1E to the IF Filter Bandwidth register
SpiWriteRegister(0x20, 0xD0);     //write 0xD0 to the Clock Recovery Oversampling Ratio register
SpiWriteRegister(0x21, 0x00);     //write 0x00 to the Clock Recovery Offset 2 register
SpiWriteRegister(0x22, 0x9D);     //write 0x9D to the Clock Recovery Offset 1 register
SpiWriteRegister(0x23, 0x49);     //write 0x49 to the Clock Recovery Offset 0 register
SpiWriteRegister(0x24, 0x00);     //write 0x00 to the Clock Recovery Timing Loop Gain 1 register
SpiWriteRegister(0x25, 0x24);     //write 0x24 to the Clock Recovery Timing Loop Gain 0 register
SpiWriteRegister(0x1D, 0x40);     /write 0x40 to the AFC Loop Gearshift Override register
```

In Si4431 revision A0 and Si443x revision B1, there are dedicated registers for setting the AFC Limit. It makes the radio easier to use; there is no need to change the Frequency Deviation register as is required with the Si4432:

```
SpiWriteRegister(0x2A, 0x20);     //write 0x20 to the AFC Limiter register
```

If the Si4431, revision A0 is used, and the application is not time-synchronized (i.e., the receiver is enabled continuously since it does not know when the packet will arrive). Then, the Modem Test register has to be programmed to a different value than its default. If the recommended register change is not applied, the receiver can stop receiving and stays in receive mode until the microprocessor issues a reset to the radio.

```
//set the Modem test register -- IMPORTANT for Si4431 revA0!
SpiWriteRegister(0x56, 0xC1);     //write 0xC1 to the Modem test register
```

Note: This setting is not required for subsequent versions of the radio.

4.1.4.4. Set Packet Configuration

The EZRadioPRO devices support various packet configurations:

- The preamble is programmable in nibbles up to 256 bytes.
- The length and the pattern of the synchron word is programmable up to 4 bytes.
- Up to 4 bytes of header can be used, with header filters on the receiver side.
- Fixed and variable length packets can be received.
- Up to 64 bytes of payload can be received.
- The chip automatically calculates and checks the CRC-16 with three different types of 16-bit CRC polynomials supported.

If the application uses a packet configuration that fits in the above options, the receive packet handler can be configured to qualify and receive the packet automatically.

The code section in this paragraph shows how the packet handler must be configured for the packet shown in Table 1 on page 9.

4.1.4.5. Tips

If the packet has a header field (up to 4 bytes), then the receive packet handler can perform header filtering for these bytes. This is a useful feature if network operation is implemented and, for example, a node just wants to receive packets from a dedicated node or group of nodes.

There are expected receive header (Check Header 3 … 0 registers) and header mask registers (Header Enable 3 … 0 registers) defined for each header byte individually. Each header byte can also be compared against the broadcast address (0xFF). If any of the received header bytes fails the filter check, the “header” bit in the Device Status register is set. Figure 8 shows the relation between the different registers and how the header filtering works.

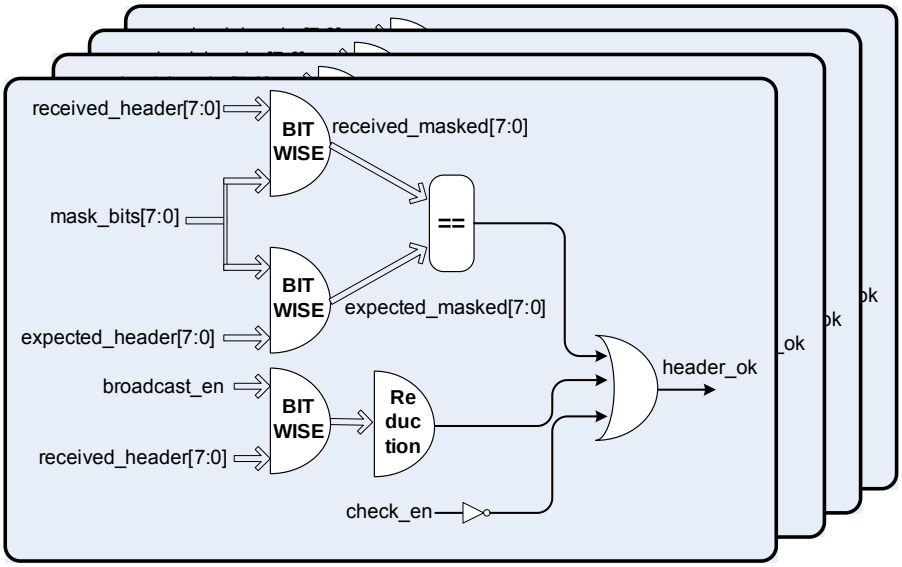


Figure 8. Receive Header Filter

This example code supports Antenna Diversity and non-Antenna diversity modes. The antenna diversity selection is made by using the precompiling definitions, SEPARATE_RX_TX, ANTENNA_DIVERSITY, and ONE_SMA_WITH_RF_SWITCH, as described in "2. Hardware Options" on page 1. The following code section configures the receive packet handler for the proper packet configuration:

```
                /*Configure the receive packet handler*/
        //Disable header bytes; set variable packet length (the length of the payload is defined by the
        //received packet length field of the packet); set the synch word to two bytes long
        SpiWriteRegister(0x33, 0x02 );      //write 0x02 to the Header Control2 register
        //Disable the receive header filters
        SpiWriteRegister(0x32, 0x00 );      //write 0x00 to the Header Control1 register

        //Set the sync word pattern to 0x2DD4
        SpiWriteRegister(0x36, 0x2D);       //write 0x2D to the Sync Word 3 register
        SpiWriteRegister(0x37, 0xD4);       //write 0xD4 to the Sync Word 2 register

        //Enable the receive packet handler and CRC-16 (IBM) check
        SpiWriteRegister(0x30, 0x85);       //write 0x85 to the Data Access Control register
        //Enable FIFO mode and GFSK modulation
        SpiWriteRegister(0x71, 0x63);       //write 0x63 to the Modulation Mode Control 2 register
        //set preamble detection threshold to 20bits
        SpiWriteRegister(0x35, 0x28);       //write 0x28 to the Preamble Detection Control  register

#ifdef ANTENNA_DIVERSITY
        //Enable antenna diversity mode
        SpiWriteRegister(0x08, 0x80);       //write 0x80 to the Operating Function Control 2 register
#endif

                /*set the GPIO's according the testcard type*/
#ifdef ANTENNA_DIVERSITY
        SpiWriteRegister(0x0C, 0x17);       //write 0x17 to the GPIO1 Configuration(set the Antenna 1 Switch used for
antenna diversity )
        SpiWriteRegister(0x0D, 0x18);       //write 0x18 to the GPIO2 Configuration(set the Antenna 2 Switch used for
antenna diversity )
#endif
#ifdef ONE_SMA_WITH_RF_SWITCH
        SpiWriteRegister(0x0C, 0x12);       //write 0x12 to the GPIO1 Configuration(set the TX state)
        SpiWriteRegister(0x0D, 0x15);       //write 0x15 to the GPIO2 Configuration(set the RX state)
#endif
```

4.1.4.6. Tips

Both RX and TX packet handler can be set to transmit and receive fixed-length packets. In this case, the length of the payload is not included in the transmitted packet but is defined by the Transmit Packet Length register. Note that the “fixpklen” bit in the Header Control 2 register must be set for this operation.

4.1.4.7. Special Considerations for the Si4432 Revision V2

The following registers must be configured for the Si4432-V2 device to ensure optimal operation.

The VCO registers must be configured using the following values:

```
                /*set the non-default Si4432 registers*/
        //set the VCO and PLL
        SpiWriteRegister(0x5A, 0x7F);       //write 0x7F to the VCO Current Trimming register
        SpiWriteRegister(0x58, 0x80);       //write 0x80 to the ChargepumpCurrentTrimmingOverride register
        SpiWriteRegister(0x59, 0x40);       //write 0x40 to the Divider Current Trimming register
```

To get the best receiver performance, it is highly recommended to set the AGC and ADC parameters according the following:

```
//set the AGC
SpiWriteRegister(0x6A, 0x0B);      //write 0x0B to the AGC Override 2 register
//set ADC reference voltage to 0.9V
SpiWriteRegister(0x68, 0x04);      //write 0x04 to the Deltasigma ADC Tuning 2 register
SpiWriteRegister(0x1F, 0x03);      //write 0x03 to the Clock Recovery Gearshift Override register
```

Note: These settings are not needed for later chip revisions, such as Si4431 revision A0.

Set the Crystal Load Capacitance register (see "3.1.7. Crystal Oscillator Tuning Capacitor" on page 16 for more details):

```
                                   //set  Crystal Oscillator Load Capacitance register
SpiWriteRegister(0x09, 0xD7);      //write 0xD7 to the Crystal Oscillator Load Capacitance register
```

4.1.4.8. Special Consideration for the Si4431 Revision A0

The following registers must be configured for the Si4431-A0 device to ensure optimal current consumption:

```
      //set VCO and PLL
      SpiWriteRegister(0x57, 0x01);//write 0x01 to the Chargepump Test register
      SpiWriteRegister(0x59, 0x00);//write 0x00 to the Divider Current Trimming
register
      SpiWriteRegister(0x5A, 0x01); //write 0x01 to the VCO Current Trimming
register
```

Note: These settings are not required for later chip revisions.

Set the Crystal Load Capacitance register (see "3.1.7. Crystal Oscillator Tuning Capacitor" on page 16 for more details):

```
//set  Crystal Oscillator Load Capacitance register
SpiWriteRegister(0x09, 0xD7);      //write 0xD7 to the Crystal Oscillator Load Capacitance register
```

4.1.4.9. Special Considerations for the Si443x revision B1

If antenna diversity is not used, the SGI bit in the AGC OVerride 1 register must be set for normal AGC operation.

However, if the antenna diversity is used, it is recommended to clear this bit. It will prevent a huge gain increase in the AGC in case the chip finds a much weaker signal on the second antenna. In this case, the first antenna will be selected for packet reception; the AGC will not be able to reduce the high gain (caused by the second antenna), and the ADC will be overridden.

```
#ifndef ANTENNA_DIVERSITY

 //set AGC Override1 Register

 SpiWriteRegister(0x69, 0x60);                    //write 0x60 to the AGC Override1
register

#endif
```

Set the Crystal Load Capacitance register (see "3.1.7. Crystal Oscillator Tuning Capacitor" on page 16 for more details):

```
//set  Crystal Oscillator Load Capacitance register

SpiWriteRegister(0x09,  0xD7);  //write  0xD7  to  the  Crystal  Oscillator  Load
Capacitance register
```

4.2. Receiving a Packet with Si4432-V2 or Si443x-B1

After initializing the MCU and the radio, the demo switches to receive mode while waiting for packets from a transmit node. Using the built-in receive packet handler, the radio receives the packet and generates an interrupt for the microcontroller only if a valid packet, or a packet with an incorrect CRC, has been received.

```
/*enable receiver chain*/
SpiWriteRegister(0x07, 0x05);//write 0x05 to the Operating Function Control 1 register
//Enable two interrupts:
// a) one which shows that a valid packet received: 'ipkval'
// b) second shows if the packet received with incorrect CRC: 'icrcerror'
SpiWriteRegister(0x05, 0x03); //write 0x03 to the Interrupt Enable 1 register
SpiWriteRegister(0x06, 0x00); //write 0x00 to the Interrupt Enable 2 register
//read interrupt status registers to release all pending interrupts
ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

/*MAIN Loop*/
while(1)
{
    //wait for the interrupt event
       if( NIRQ == 0 )
       {
```

When an interrupt is generated by the radio, the MCU must determine the cause of the interrupt by reading the interrupt status registers. If a packet is received with an incorrect CRC, the demo stops the receiver, blinks all the LEDs (showing the error for the user), and restarts the receiver chain.

```
//read interrupt status registers
ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

/*CRC Error interrupt occured*/
if( (ItStatus1 & 0x01) == 0x01 )
{
        //disable the receiver chain
        SpiWriteRegister(0x07, 0x01);//write 0x01 to the Operating Function Control 1 register
        //reset the RX FIFO
        SpiWriteRegister(0x08, 0x02);//write 0x02 to the Operating Function Control 2 register
        SpiWriteRegister(0x08, 0x00);//write 0x00 to the Operating Function Control 2 register
        //blink all LEDs to show the error
        LED1 = 1;
        LED2 = 1;
        LED3 = 1;
        LED4 = 1;
        for(delay = 0; delay < 10000;delay++);
        LED1 = 0;
        LED2 = 0;
        LED3 = 0;
        LED4 = 0;
        //enable the receiver chain again
        SpiWriteRegister(0x07, 0x05);//write 0x05 to the Operating Function Control 1 register
}
```

If a valid packet with a correct CRC is received, the MCU reads the length of the received payload, checks that the allocated buffer in the MCU is large enough to store the packet, and reads the content of the RX FIFO. It then blinks the appropriate LED according to the content of the packet.

If the payload cannot fit into the allocated buffer of the MCU or the payload is different than the expected packet(s), the MCU drops the packet and restarts the receiver chain:

```
                        /*packet received interrupt occurred*/
                        if( (ItStatus1 & 0x02) == 0x02 )
                        {
                                //disable the receiver chain
                                SpiWriteRegister(0x07, 0x01);//write 0x01 to the Operating Function Control 1 register
                                //Read the length of the received payload
                                length = SpiReadRegister(0x4B);//read the Received Packet Length register
                                //check whether the received payload is not longer than the allocated buffer in the MCU
                                if(length < 11)
                                {
                                        //Get the received payload from the RX FIFO
                                        for(temp8=0;temp8 < length;temp8++)
                                        {
                                                payload[temp8] = SpiReadRegister(0x7F);//read the FIFO Access register
                                        }

                                        //check whether the content of the packet is what the demo expects
                                        if( length == 8 )
                                        {
                                                if( memcmp(&payload[0], "BUTTON1", 7) == 0 )
                                                {
                                                //blink LED1 to show that the packet received
                                                        LED1 = 1;
                                                        for(delay = 0; delay < 10000;delay++);
                                                        LED1 = 0;
                                                }
                                                if( memcmp(&payload[0], "BUTTON2", 7) == 0 )
                                                {
                                                //blink LED2 to show that the packet received
                                                        LED2 = 1;
                                                        for(delay = 0; delay < 10000;delay++);
                                                        LED2 = 0;
                                                }
                                                if( memcmp(&payload[0], "BUTTON3", 7) == 0 )
                                                {
                                                //blink LED3 to show that the packet received
                                                        LED3 = 1;
                                                        for(delay = 0; delay < 10000;delay++);
                                                        LED3 = 0;
                                                }
                                                if( memcmp(&payload[0], "BUTTON4", 7) == 0 )
                                                {
                                                //blink LED4 to show that the packet received
                                                        LED4 = 1;
                                                        for(delay = 0; delay < 10000;delay++);
                                                        LED4 = 0;
                                                }
                                        }
                                }
                                //reset the RX FIFO
                                SpiWriteRegister(0x08, 0x02);//write 0x02 to the Operating Function Control 2 register
                                SpiWriteRegister(0x08, 0x00);//write 0x00 to the Operating Function Control 2 register
                                //enable the receiver chain again
                                SpiWriteRegister(0x07, 0x05);//write 0x05 to the Operating Function Control 1 register
                        }
                }
        }
}
```

Note: The source code compiled for the EZLink fast prototyping platform accepts packets with “Button1” payload only.

4.2.1. Receiving a Packet with the Si4431 Revision A0

If the Si4431 revision A0 is used, there is an errata that can cause the radio to stall in the synchronization word detection state if a false preamble detection occurs. When the radio stalls in the synchronization word detection state, it can result in an increase in the packet error rate. This problem can be corrected by either increasing the preamble detection threshold, reducing the probability of a false preamble detection, or applying the following software workaround:

- Enable the Valid Preamble Detect and Synch Word Detect interrupts. The radio then begins waiting for the preamble to be detected.
- If the chip detects the preamble, it will generate an interrupt (ipreaval) to the MCU. Run a synchron word timeout timer in the microcontroller while the chip is waiting for the synchron word. The timeout duration should be (preamble- preamble detection threshold + synchron word) * byte interval. If the timer expires without receiving the synchron word, then disable and enable the receiver.
- After a successful synchron word detection, the procedure is the same as the Si4432-V2 or Si443x-B1, i.e. the packet valid and the CRC error interrupt should be enabled, and the MCU has to wait for these interrupts.

The code section, which is different than the Si4432, is shown below:

```
/*enable receiver chain*/
SpiWriteRegister(0x07, 0x05);//write 0x05 to the Operating Function Control 1 register
//Enable two interrupts:
// a) one wich shows that a valid preamble received:'ipreaval'
// b) second shows if a sync word received:'iswdet'
SpiWriteRegister(0x05, 0x00); //write 0x00 to the Interrupt Enable 1 register
SpiWriteRegister(0x06, 0xC0); //write 0xC0 to the Interrupt Enable 2 register
//read interrupt status registers to release all pending interrupts
ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

/*MAIN Loop*/
while(1)
{
        //wait for the interrupt event
        if( NIRQ == 0 )
        {
                //read interrupt status registers
                ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

                //check whether preamble is detected
                if( (ItStatus2 & 0x40) == 0x40 )
                {//preamble detected
                        //wait for the synch word interrupt with timeout -- THIS is the proposed SW workaround
                        //start a timer in the MCU and during timeout check whether synch word interrupt
                        happened or not
                        delay = 0;
                        do {
                                delay++;
                        } while((delay < 20000) && (NIRQ == 1));

                        //check whether the synch word interrupt is detected
```

```
if( NIRQ == 0)
{//synch word detected correctly
        //read interrupt status registers
        ItStatus1 = SpiReadRegister(0x03); //read the Interrupt Status1 register
        ItStatus2 = SpiReadRegister(0x04); //read the Interrupt Status2 register
        //Enable two interrupts:
        // a) one which shows that a valid packet received: 'ipkval'
        // b) second shows if the packet received with incorrect CRC: 'icrcerror'
        SpiWriteRegister(0x05, 0x03);  //write 0x03 to the Interrupt Enable 1 register
        SpiWriteRegister(0x06, 0x00);  //write 0x00 to the Interrupt Enable 2 register
        //read interrupt status registers to release all pending interrupts
        ItStatus1 = SpiReadRegister(0x03); //read the Interrupt Status1 register
        ItStatus2 = SpiReadRegister(0x04); //read the Interrupt Status2 register
        //wait for the interrupt event
        while(NIRQ == 1);
        //read interrupt status registers
        ItStatus1 = SpiReadRegister(0x03); //read the Interrupt Status1 register
        ItStatus2 = SpiReadRegister(0x04); //read the Interrupt Status2 register

        /*CRC Error interrupt occured*/
        if( (ItStatus1 & 0x01) == 0x01 )
        {
                //disable the receiver chain
                SpiWriteRegister(0x07, 0x01); //write 0x01 to the Operating Function
                Control 1 register
                //blink all LEDs to show the error
                RX_LED = 1;
                TX_LED = 1;
                for(delay = 0; delay < 10000;delay++);
                RX_LED = 0;
                TX_LED = 0;
        }

        /*packet received interrupt occured*/
        if( (ItStatus1 & 0x02) == 0x02 )
        {
                //disable the receiver chain
                SpiWriteRegister(0x07, 0x01); //write 0x01 to the Operating Function
                Control 1 register
                //Read the length of the received payload
                length = SpiReadRegister(0x4B); //read the Received Packet Length
                register
                //check whether the received payload is not longer than the allocated
                buffer in the MCU
                if(length < 11)
                {
                        //Get the reeived payload from the RX FIFO
                        for(temp8=0;temp8 < length;temp8++)
                        {
```

```
                                        payload[temp8] = SpiReadRegister(0x7F);
                                    }
                                    //check whether the content of the packet is what the demo
                                    expects
                                    if( length == 8 )
                                    {
                                                                        if(
                                        memcmp(&payload[0], "BUTTON1", 7) == 0 )
                                        {//blink LED1 to show that the packet received
                                        RX_LED = 1;
                                        for(delay = 0; delay < 10000;delay++);
                                        RX_LED = 0;
                                    }
                                    }
                                }
                            }
                            //Enable two interrupts agin:
                            // a) one wich shows that a valid preamble received:'ipreaval'
                            // b) second shows if a sync word received:'iswdet'
                            SpiWriteRegister(0x05, 0x00); //write 0x00 to the Interrupt Enable 1 register
                            SpiWriteRegister(0x06, 0xC0); //write 0xC0 to the Interrupt Enable 2 register
                            //read interrupt status registers to release all pending interrupts
                            ItStatus1 = SpiReadRegister(0x03); //read the Interrupt Status1 register
                            ItStatus2 = SpiReadRegister(0x04); //read the Interrupt Status2 register
                            //enable the receiver chain again
                            SpiWriteRegister(0x07, 0x05); //write 0x05 to the Operating Function Control 1
                            register
                        }
                        else
                        {//synch word is not detected within timeout --> reset RX chain to workaround the
                        Si4431-A0 errata
                            //disable receiver chain
                            SpiWriteRegister(0x07, 0x01); //write 0x01 to the Operating Function Control 1
                            register
                            //read interrupt status
                            ItStatus2 = SpiReadRegister(0x04); //read the Interrupt Status2 register
                            //enable RX chain
                            SpiWriteRegister(0x07, 0x05); //write 0x05 to the Operating Function Control 1
                            register
                        }
                    }
                }
            }
        }
}
```

5. Bidirectional Communication—Send Acknowledgement

The software example realizes a bidirectional, half-duplex communication protocol between two nodes. The same firmware has to be loaded into both devices.

After initializing the microcontroller and the EZRadioPRO transceiver, the software goes into continuous receive mode. It then waits for a packet to be received or for the user to press the push button.

If a valid packet is received with a payload of “Button1” the software blinks “LED1” on the software development board or the RX LED on the EZLink fast prototyping platform to indicate a successful packet reception. It forms and sends an acknowledgement packet back to the originator. After successful packet transmission, it goes back into continuous receive mode.

If Push Button 1 is pressed, the software disables the receiving mode, forms a packet with a payload of “Button1”, and transmits the packet. During packet transmission, it blinks “LED2” on the software development board or “TX LED” on the EZLink platform. After successful packet transmission, it goes back into continuous receive mode and waits for the acknowledgement. If the node receives the acknowledgement packet, it blinks “LED1” or “RX LED” and goes back into receive mode.

5.1. Software Flow Chart

Figure 9 shows the flow chart for the demo application:

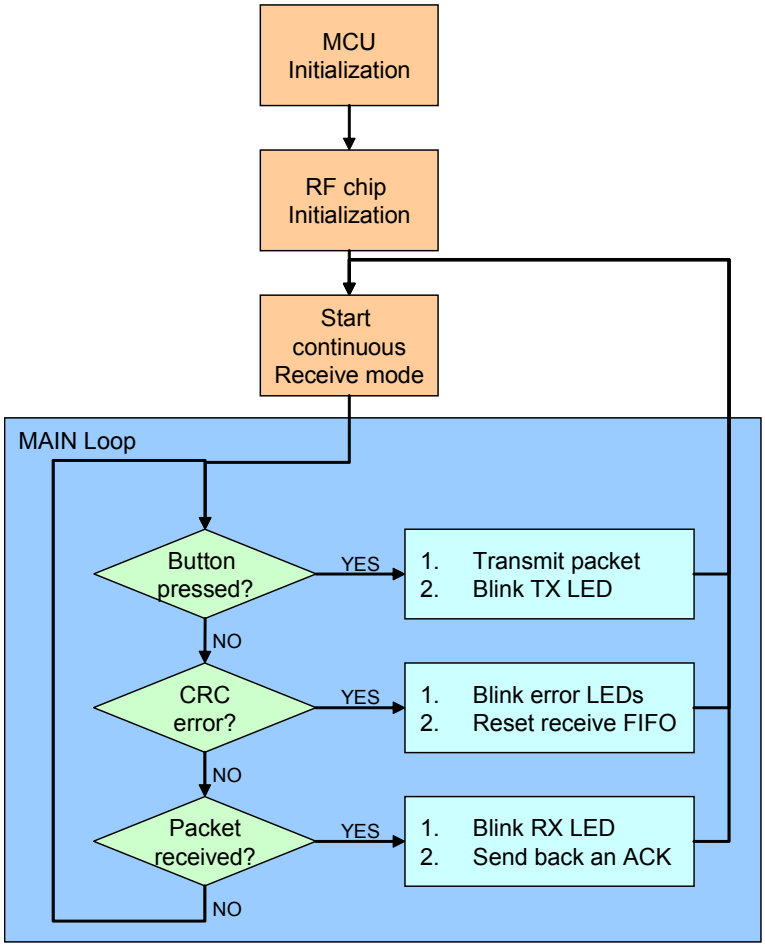


Figure 9. Flow Chart

5.2. Software Implementation

The software example uses the built-in receive and transmit packet handler and FIFO of the transceiver chip. The code transmits and receives packets with the same packet format shown in Table 1 on page 9. Also, the modulation method is GFSK; the data rate is 9.6 kbps, and the transmit deviation is ±45 kHz. These are the same RF parameters used in the earlier examples. For bidirectional communication, both the transmitter and receiver portion of the transceiver chip have to be configured. The initialization code section for both transmitter and receiver operations are discussed in the earlier examples ("3. Packet Transmission Using the Transmit Packet Handler" on page 9 and "4. Packet Receiving Using the Receive Packet Handler" on page 18). These chapters highlight only those details required for bidirectional communication.

5.2.1. Interrupt Flags

The EZRadioPRO devices can provide interrupt for several events (e.g., reset occurred, crystal stabilized, packet transmitted, packet received, etc.). When the microcontroller receives an interrupt, it has to read the Interrupt Status registers to figure out what caused the interrupt. To achieve the fastest reaction time, it is recommended to enable only the relevant interrupts for each operating mode. In this case, the software does not need to always check all the status flags to identify the source of the interrupt, but the few relevant bits only. For example:

- If the software is in continuous receive mode, it is recommended to enable the packet valid (“enpkvalid” bit in the Interrupt Enable 1 register) and the CRC error (“encrcerror” bit in the Interrupt Enable 1 register) interrupts.
- If the software transmits a packet, it is sufficient to enable the packet sent interrupt (“enpksent” bit in the Interrupt Enable 1 register).

5.2.2. Frequency Deviation Register

If using the Si4432 Rev V2 transceiver chip, the Frequency Deviation register has two roles:

- During packet transmit, it defines the transmit frequency deviation of the RF link.
- During receive mode, if the AFC is enabled, this register defines the maximum frequency offset for the Auto-Frequency Calibration.

It is important to set the register for the appropriate function before going into receive or transmit mode.

5.2.3. Initialization of the EZRadioPRO Transceiver

The following code section shows the transceiver chip initialization. It sets all the registers of the radio needed for receive or transmit operation:

```
            /* ============================================================ *
             *                  Initialize the Si4432 ISM chip              *
             * ============================================================ */
//SW reset
SpiWriteRegister(0x07, 0x80);//write 0x80 to the Operating & Function Control1 register
//wait for chip ready interrupt from the radio (while the nIRQ pin is high)
while ( NIRQ == 1);
//read interrupt status registers to clear the interrupt flags and release NIRQ pin
ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register

        /*set the physical parameters*/
//set the center frequency to 915 MHz
SpiWriteRegister(0x75, 0x75);               //write 0x75 to the Frequency Band Select register
SpiWriteRegister(0x76, 0xBB);               //write 0xBB to the Nominal Carrier Frequency1 register
SpiWriteRegister(0x77, 0x80);               //write 0x80 to the Nominal Carrier Frequency0 register

//set the desired TX data rate (9.6kbps)
SpiWriteRegister(0x6E, 0x4E);               //write 0x4E to the TXDataRate 1 register
SpiWriteRegister(0x6F, 0xA5);               //write 0xA5 to the TXDataRate 0 register
SpiWriteRegister(0x70, 0x2C);               //write 0x2C to the Modulation Mode Control 1 register


        /*set the modem parameters according to the excel calculator(parameters: 9.6 kbps, deviation: 45 kHz, channel filter BW:
112.1 kHz*/
```

```
        SpiWriteRegister(0x1C, 0x05);           //write 0x05 to the IF Filter Bandwidth register
        SpiWriteRegister(0x20, 0xA1);           //write 0xA1 to the Clock Recovery Oversampling Ratio register
        SpiWriteRegister(0x21, 0x20);           //write 0x20 to the Clock Recovery Offset 2 register
        SpiWriteRegister(0x22, 0x4E);           //write 0x4E to the Clock Recovery Offset 1 register
        SpiWriteRegister(0x23, 0xA5);           //write 0xA5 to the Clock Recovery Offset 0 register
        SpiWriteRegister(0x24, 0x00);           //write 0x00 to the Clock Recovery Timing Loop Gain 1 register
        SpiWriteRegister(0x25, 0x13);           //write 0x13 to the Clock Recovery Timing Loop Gain 0 register
        SpiWriteRegister(0x1D, 0x40);           //write 0x40 to the AFC Loop Gearshift Override register
        SpiWriteRegister(0x72, 0x1F);           //write 0x1F to the Frequency Deviation register


                /*set the packet structure and the modulation type*/
        //set the preamble length to 10bytes if the antenna diversity is used and set to 5bytes if not
#ifdef ANTENNA_DIVERSITY
        SpiWriteRegister(0x34, 0x18);           //write 0x18 to the Preamble Length register
#else
        SpiWriteRegister(0x34, 0x0C);           //write 0x0C to the Preamble Length register
#endif
        //set preamble detection threshold to 20bits
        SpiWriteRegister(0x35, 0x28);           //write 0x28 to the Preamble Detection Control  register

        //Disable header bytes; set variable packet length (the length of the payload is defined by the
        //received packet length field of the packet); set the synch word to two bytes long
        SpiWriteRegister(0x33, 0x02);           //write 0x02 to the Header Control2 register

        //Set the sync word pattern to 0x2DD4
        SpiWriteRegister(0x36, 0x2D);           //write 0x2D to the Sync Word 3 register
        SpiWriteRegister(0x37, 0xD4);           //write 0xD4 to the Sync Word 2 register

        //enable the TX & RX packet handler and CRC-16 (IBM) check
        SpiWriteRegister(0x30, 0x8D);           //write 0x8D to the Data Access Control register
        //Disable the receive header filters
        SpiWriteRegister(0x32, 0x00 );          //write 0x00 to the Header Control1 register
        //enable FIFO mode and GFSK modulation
        SpiWriteRegister(0x71, 0x63);           //write 0x63 to the Modulation Mode Control 2 register

#ifdef ANTENNA_DIVERSITY
        //enable the antenna diversity mode
        SpiWriteRegister(0x08, 0x80);//write 0x80 to the Operating Function Control 2 register
#endif

                /*set the GPIO's according the testcard type*/
#ifdef ANTENNA_DIVERSITY
        SpiWriteRegister(0x0C, 0x17);           //write 0x17 to the GPIO1 Configuration(set the Antenna 1 Switch used for
antenna diversity )
        SpiWriteRegister(0x0D, 0x18);           //write 0x18 to the GPIO2 Configuration(set the Antenna 2 Switch used for
antenna diversity )
#endif
#ifdef ONE_SMA_WITH_RF_SWITCH
        SpiWriteRegister(0x0C, 0x12);           //write 0x12 to the GPIO1 Configuration(set the TX state)
        SpiWriteRegister(0x0D, 0x15);           //write 0x15 to the GPIO2 Configuration(set the RX state)
#endif

                /*set the non-default Si4432 registers*/
        //set the VCO and PLL
        SpiWriteRegister(0x5A, 0x7F);           //write 0x7F to the VCO Current Trimming register
        SpiWriteRegister(0x58, 0x80);           //write 0x80 to the ChargepumpCurrentTrimmingOverride register
        SpiWriteRegister(0x59, 0x40);           //write 0x40 to the Divider Current Trimming register
        //set the AGC
        SpiWriteRegister(0x6A, 0x0B);           //write 0x0B to the AGC Override 2 register
        //set ADC reference voltage to 0.9V
        SpiWriteRegister(0x68, 0x04);           //write 0x04 to the Deltasigma ADC Tuning 2 register
        SpiWriteRegister(0x1F, 0x03);           //write 0x03 to the Clock Recovery Gearshift Override register
        //set  Crystal Oscillator Load Capacitance register
        SpiWriteRegister(0x09, 0xD7);           //write 0xD7 to the Crystal Oscillator Load Capacitance register
```

5.2.4. Receiving Packets

After configuring the MCU and the radio, the demo goes into continuous receive mode while waiting for packets from a node or a push button input.

```
        /*enable receiver chain*/
SpiWriteRegister(0x07, 0x05);           //write 0x05 to the Operating Function Control 1 register
//Enable two interrupts:
// a) one which shows that a valid packet received: 'ipkval'
// b) second shows if the packet received with incorrect CRC: 'icrcerror'
SpiWriteRegister(0x05, 0x03);           //write 0x03 to the Interrupt Enable 1 register
SpiWriteRegister(0x06, 0x00);           //write 0x00 to the Interrupt Enable 2 register
//read interrupt status registers to release all pending interrupts
ItStatus1 = SpiReadRegister(0x03);      //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);      //read the Interrupt Status2 register
```

The MCU polls the push button and the interrupt pin of the radio in the main loop.

If the *push button is pressed*, the demo disables the receiver mode, forms a data packet (with payload of "Button1"), and transmits it to the other node:

```
/*MAIN Loop*/
while(1)
{
        //Poll the port pins of the MCU to figure out whether the push button is pressed or not
        if(PB1 == 0)
        {
                //Wait for releasing the push button
                while( PB1 == 0 );
                //disable the receiver chain (but keep the XTAL running to have shorter TX on time!)
                SpiWriteRegister(0x07, 0x01);   //write 0x01 to the Operating Function Control 1 register
                //turn on the LED to show the packet transmission
                LED1 = 1;
                //The Tx deviation register has to set according to the deviation before every transmission (+-
45kHz)
                SpiWriteRegister(0x72, 0x48);   //write 0x48 to the Frequency Deviation register
                /*SET THE CONTENT OF THE PACKET*/
                //set the length of the payload to 8bytes
                SpiWriteRegister(0x3E, 8);      //write 8 to the Transmit Packet Length register
                //fill the payload into the transmit FIFO
                SpiWriteRegister(0x7F, 0x42);   //write 0x42 ('B') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x55);   //write 0x55 ('U') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x54);   //write 0x54 ('T') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x54);   //write 0x54 ('T') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x4F);   //write 0x4F ('O') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x4E);   //write 0x4E ('N') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x31);   //write 0x31 ('1') to the FIFO Access register
                SpiWriteRegister(0x7F, 0x0D);   //write 0x0D (CR) to the FIFO Access register

                //Disable all other interrupts and enable the packet sent interrupt only.
                //This will be used for indicating the successfull packet transmission for the MCU
                SpiWriteRegister(0x05, 0x04);   //write 0x04 to the Interrupt Enable 1 register
                SpiWriteRegister(0x06, 0x00);   //write 0x03 to the Interrupt Enable 2 register
                //Read interrupt status regsiters. It clear all pending interrupts and the nIRQ pin goes back to high.
                ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

                /*enable transmitter*/
                //The radio forms the packet and send it automatically.
```

```
                SpiWriteRegister(0x07, 0x09);//write 0x09 to the Operating Function Control 1 register

                /*wait for the packet sent interrupt*/
                //The MCU just needs to wait for the 'ipksent' interrupt.
                while(NIRQ == 1);
                //read interrupt status registers to release the interrupt flags
                ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register

                //wait a bit for showing the LED a bit longer
                for(delay = 0; delay < 10000;delay++);
                //turn off the LED
                LED1 = 0;

                //after packet transmission set the interrupt enable bits according receiving mode
                //Enable two interrupts:
                // a) one which shows that a valid packet received: 'ipkval'
                // b) second shows if the packet received with incorrect CRC: 'icrcerror'
                SpiWriteRegister(0x05, 0x03);               //write 0x03 to the Interrupt Enable 1 register
                SpiWriteRegister(0x06, 0x00);               //write 0x00 to the Interrupt Enable 2 register
                //read interrupt status registers to release all pending interrupts
                ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register
                //set the Frequency Deviation register according to the AFC limiter
                SpiWriteRegister(0x72, 0x1F);               //write 0x1F to the Frequency Deviation register

                /*enable receiver chain again*/
                SpiWriteRegister(0x07, 0x05);           //write 0x05 to the Operating Function Control 1 register
        }
```

Using the built-in receive packet handler, the radio *receives the packet* and generates an interrupt for the MCU only if a valid packet or a packet with incorrect CRC is received. Upon receiving an interrupt, the microcontroller checks the cause of the interrupt by reading the Interrupt Status registers:

- If a packet is received with an incorrect CRC, the demo drops the packet, resets the receive FIFO, and goes back into receive mode.
- If a packet is received with a payload of “Button1,” the demo blinks the RX LED and sends back an acknowledgement packet to the originator.
- If an acknowledgement packet is received, the demo blinks the RX LED showing the successful bidirectional packet transmission.

```
//wait for the interrupt event
        //If it occurs, then it means a packet received or CRC error happened
        if( NIRQ == 0 )
        {
                //disable the receiver chain
                SpiWriteRegister(0x07, 0x01);       //write 0x01 to the Operating Function Control 1 register
                //read interrupt status registers
                ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

                /*CRC Error interrupt occured*/
                if( (ItStatus1 & 0x01) == 0x01 )
                {           //reset the RX FIFO
                            SpiWriteRegister(0x08, 0x02);//write 0x02 to the Operating Function Control 2 register
                            SpiWriteRegister(0x08, 0x00);//write 0x00 to the Operating Function Control 2 register
                            //blink all LEDs to show the error
```

```
                LED1 = 1;
                LED2 = 1;
                LED3 = 1;
                LED4 = 1;
                for(delay = 0; delay < 10000;delay++);
                LED1 = 0;
                LED2 = 0;
                LED3 = 0;
                LED4 = 0;
        }

        /*packet received interrupt occured*/
        if( (ItStatus1 & 0x02) == 0x02 )
        {
                //Read the length of the received payload
                length = SpiReadRegister(0x4B); //read the Received Packet Length register
                //check whether the received payload is not longer than the allocated buffer in the MCU
                if(length < 11)
                {
                        //Get the reeived payload from the RX FIFO
                        for(temp8=0;temp8 < length;temp8++)
                        {
                                payload[temp8] = SpiReadRegister(0x7F);//read the FIFO Access
register

                        }

                        //check whether the acknowledgement packet received
                        if( length == 4 )
                {
                if( memcmp(&payload[0], "ACK", 3) == 0 )
        {
                   //blink LED2 to show that ACK received
                LED2 = 1;
                for(delay = 0; delay < 10000;delay++);
                LED2 = 0;
        }
                        }

                        //check whether an expected packet received, this should be acknowledged
                        if( length == 8 )
                        {
                                if( memcmp(&payload[0], "BUTTON1", 7) == 0 )
                                {
                                //blink LED2 to show that the packet received
                                        LED2 = 1;
                                        for(delay = 0; delay < 10000;delay++);
                                        LED2 = 0;

                                        /*send back an acknowledgement*/
                                        //turn on LED1 to show packet transmission
                                        LED1 = 1;
                                        //The Tx deviation register has to set according to the deviation
before

                                        //every transmission (+-45kHz)
                                        //write 0x48 to the Frequency Deviation register
                                        SpiWriteRegister(0x72, 0x48);
                                        /*set packet content*/
                                        //set the length of the payload to 4bytes
                                        //write 4 to the Transmit Packet Length register
```

```
                                        SpiWriteRegister(0x3E, 4);
                                        //fill the payload into the transmit FIFO
                                        //write A to the FIFO Access register
                                        SpiWriteRegister(0x7F, 0x41);
//write C to the FIFO Access register
                                        SpiWriteRegister(0x7F, 0x43);
//write K to the FIFO Access register
                                        SpiWriteRegister(0x7F, 0x4B);
//write CR to the FIFO Access register
                                        SpiWriteRegister(0x7F, 0x0D);
                                        //Disable all other interrupts and enable the packet sent
interrupt only.
                                        //This will be used for indicating the successfull packet
transmission for
                                        //the MCU
                                        //write 0x04 to the Interrupt Enable 1 register
                                        SpiWriteRegister(0x05, 0x04);
//write to the Interrupt Enable 2 register
                                        SpiWriteRegister(0x06, 0x00);
//Read interrupt status regsiters. It clear all pending interrupts and the
                                        //nIRQ pin goes back to high.
                                        //read the Interrupt Status1 register
ItStatus1 = SpiReadRegister(0x03);
                                        //read the Interrupt Status2 register
                                        ItStatus2 = SpiReadRegister(0x04);
                                        /*enable transmitter*/
                                        //The radio forms the packet and send it automatically.
                                        //write 0x09 to the Operating Function Control 1 register
                                        SpiWriteRegister(0x07, 0x09);
                                        /*wait for the packet sent interrupt*/
                                        //The MCU just needs to wait for the 'ipksent' interrupt.
                                        while(NIRQ == 1);
                                        //read interrupt status registers to release the interrupt flags
                                        //read the Interrupt Status1 register
                                        ItStatus1 = SpiReadRegister(0x03);
//read the Interrupt Status2 register
                                        ItStatus2 = SpiReadRegister(0x04);
                                        //wait a bit for showing the LED a bit longer
                                        for(delay = 0; delay < 10000;delay++);
                                        //turn off the LED
                                        LED1 = 0;

                                        //after packet transmission set the interrupt enable bits
according
                                        //receiving mode
                                        //Enable two interrupts:
                                        // a) one which shows that a valid packet received: 'ipkval'
                                        // b) second shows if the packet received with incorrect CRC:
'icrcerror'
                                        //write 0x03 to the Interrupt Enable 1 register
                                        SpiWriteRegister(0x05, 0x03);
//write 0x00 to the Interrupt Enable 2 register
                                        SpiWriteRegister(0x06, 0x00);
//read interrupt status registers to release all pending interrupts
                                        //read the Interrupt Status1 register
                                        ItStatus1 = SpiReadRegister(0x03);
//read the Interrupt Status2 register
                                        ItStatus2 = SpiReadRegister(0x04);
//set the Frequency Deviation register according to the AFC limiter
```

```
                                                    //write 0x1F to the Frequency Deviation register
                                                    SpiWriteRegister(0x72, 0x1F);
}
                                                }
                                            }
                                        }
                                        /*enable receiver chain again*/
                                        SpiWriteRegister(0x07, 0x05);//write 0x05 to the Operating Function Control 1 register
                                }
                            }
}
```

Note: The receive function for Si4431 is slightly different; see "4.2.1. Receiving a Packet with the Si4431 Revision A0" on page 29 for more details. The sample code for Si443x-B1 is slightly different since it has separate Frequency Deviation and AFC Limit Registers; so, there is no need to set them every time.

6. Transmit and Receive Long Packets Using the FIFO

The EZRadioPRO devices have 64 byte transmit and receive FIFOs. However, it is possible to transmit and receive packets with longer payloads of up to 255 bytes. The following example source code realizes bidirectional communication by transmitting and receiving packets with 128 bytes of payload.

The packet format used in this example uses the following longer packet format:

Table 3. Packet Configuration

Preamble					Synchron pattern		Payload length	Payload				CRC	
55	55	55	55	55	2D	D4	80	00	01	…	7F	16	1B
5 bytes (10 bytes if Ant. Diversity is used)					2 bytes		1 byte	128 bytes				2 bytes	

6.1. How to Send Longer than 64 bytes of Payload

The built-in transmit and receive packet handlers provide three different FIFO status signals: Transmit FIFO Almost Full, Transmit FIFO Almost Empty, and Receive FIFO Almost Full. These signals can be used by the MCU to send and receive packets that are larger than the built-in FIFOs.

The transmit FIFO has two programmable thresholds, and the radio can provide interrupts when the data in the transmit FIFO reaches these thresholds. The first threshold is the Transmit FIFO Almost Full (set by the TX FIFO Control 1 register). If the number of bytes filled into the FIFO reaches this level, the radio can provide an interrupt for the MCU to start the packet transmission. The second threshold is the Transmit FIFO Almost Empty (set by the TX FIFO Control 2 register). If the number of bytes in the transmit FIFO reaches this level, the radio can provide an interrupt for the MCU. The microcontroller must switch out of transmit mode or fill more data into the transmit FIFO.

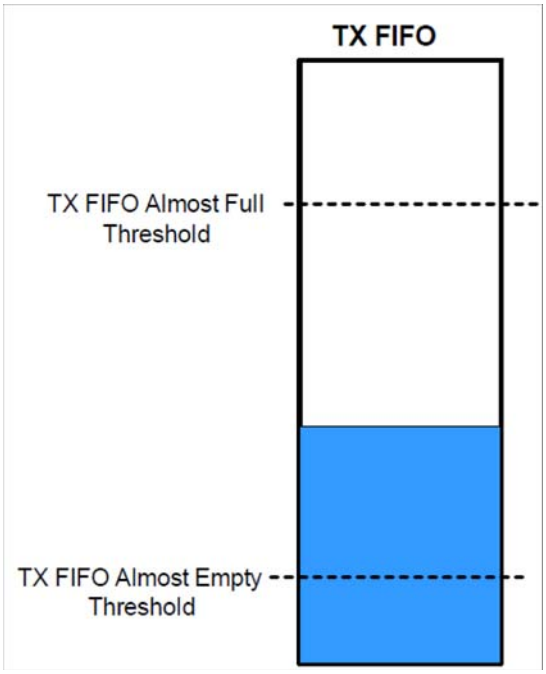


Figure 10. Transmit FIFO Status Signals

The receive FIFO has one threshold, Receive FIFO Almost Full (set by the RX FIFO Control register). If the number of received bytes stored into the receive FIFO reaches this threshold, the radio can generate an interrupt for the MCU to read the data from the FIFO.

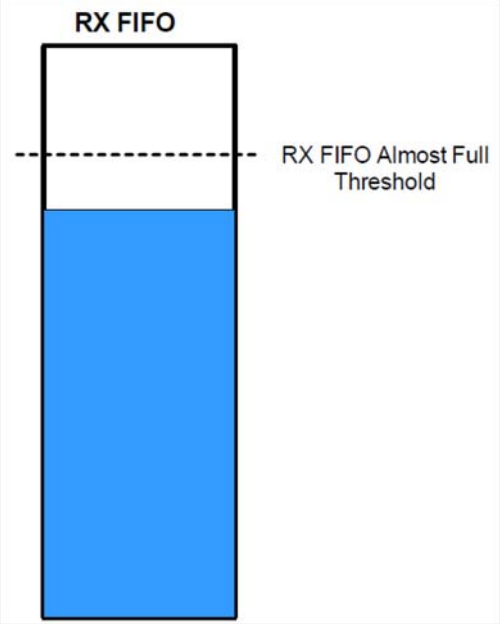


Figure 11. Receive FIFO

6.2. Software Flow Chart

The software example uses the Transmit FIFO Almost Empty and the Receive FIFO Almost empty status flags for transmit and receive packets with 128 bytes of payload in the following way:

- **Transmit a Packet**—The demo sets the Transmit FIFO Almost Empty Threshold to 10 bytes. It fills the first 64 bytes of the payload into the transmit FIFO and starts packet transmission. It then waits for the Transmit FIFO Almost Empty interrupt. When the interrupt occurs, it fills the next 32 bytes into the FIFO and waits for the Transmit FIFO Almost Empty interrupt. When the interrupt occurs, it fills the last 32 bytes of the payload into the FIFO and waits for the packet sent interrupt.
- **Receive a Packet**—The demo sets the Receive FIFO Almost Full Threshold to 54 bytes. Before receiving a valid packet interrupt, the MCU waits for the Receive FIFO Almost Full interrupt and then reads 32 bytes from the receive FIFO. The radio will repeat this process until the entire 128 byte payload is received. In the case of a CRC error, the MCU will reset the receive FIFO and discard the packet.

Figure 12 shows the software flow chart.

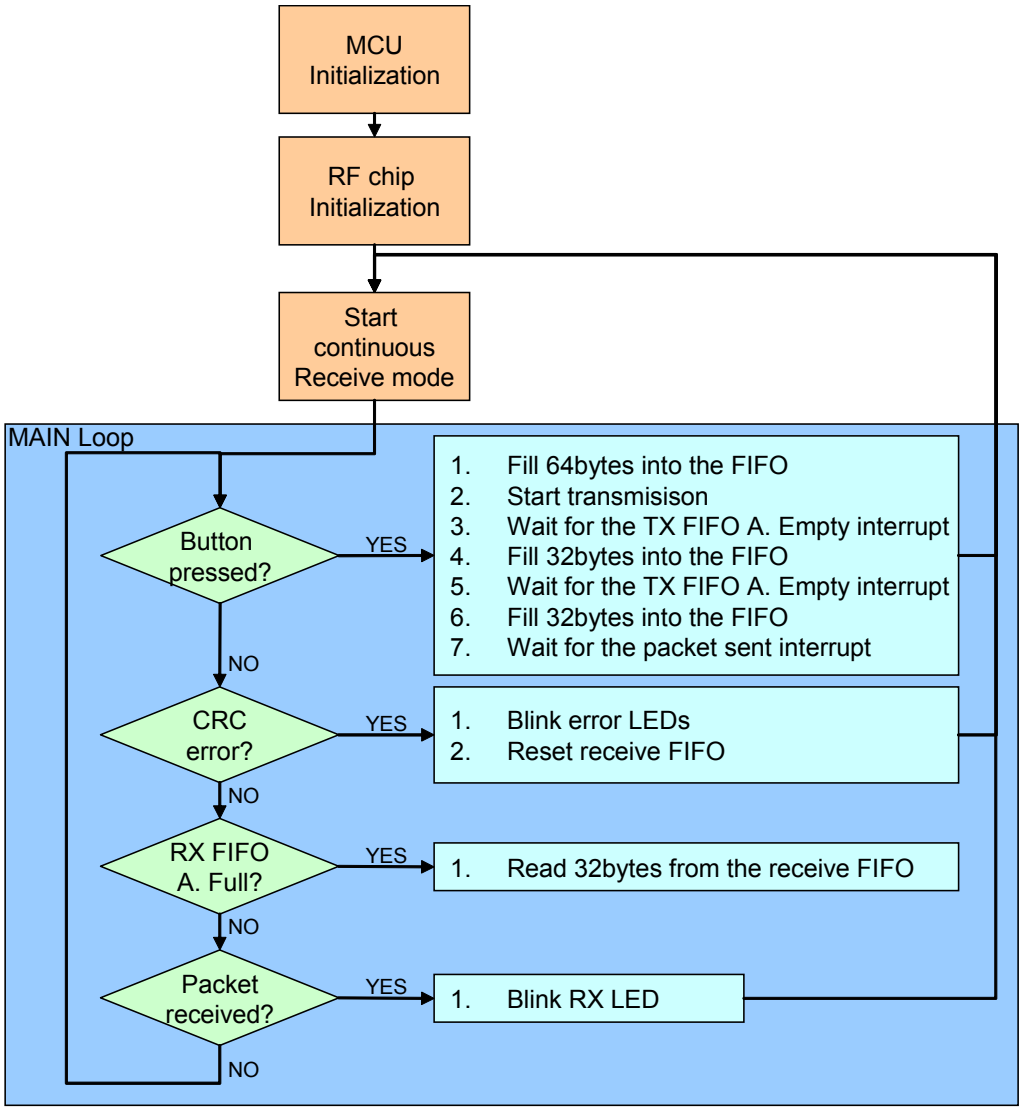


Figure 12. Demo Software Block Diagram

6.3. Software Implementation

6.3.1. Initialization of the MCU and the Radio

For details of the radio configuration, see "5.2.3. Initialization of the EZRadioPRO Transceiver" on page 33.

6.3.2. Packet Transmission

The following code segment shows how the packet transmission is realized. To simplify the software, the payload of the transmitted packet is stored in FLASH as a byte array. The declaration of the array can be found at the beginning of the C code:

```
/* ======================================================= *
 *                  GLOBAL VARIABLE       *
 * ======================================================= */


code U8 tx_packet[128] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
                          16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
                          32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
                          48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
                           64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
                           80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
                           96, 97, 98, 99, 100,101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111,
                           112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122,123, 124, 125, 126, 127};
```

If the push button is pressed, the software switches out of receive mode and fills the first 64 bytes of the packet into the transmit FIFO. The MCU starts packet transmission and waits for the Transmit FIFO Almost Empty interrupt (“itxffafull” bit in the Interrupt Status 1 register) and fills the next 32 bytes into the FIFO. It repeats this process again and waits for the packet sent interrupt (“ipksent” bit in the Interrupt Status 1 register). After successful packet transmission, it goes into continuous receive mode.

```
//Poll the port pins of the MCU to figure out whether the push button is pressed or not
if(PB1 == 0)
{
        //Wait for releasing the push button
        while( PB1 == 0 );
        //disable the receiver chain (but keep the XTAL running to have shorter TX on time!)
        SpiWriteRegister(0x07, 0x01);//write 0x01 to the Operating Function Control 1 register

        //turn on the LED to show the packet transmission
        LED1 = 1;
        //The Tx deviation register has to set according to the deviation before every transmission (+-
45kHz)
        SpiWriteRegister(0x72, 0x48);//write 0x48 to the Frequency Deviation register

        /*SET THE CONTENT OF THE PACKET*/
        //set the length of the payload to 128bytes
        SpiWriteRegister(0x3E, 128);//write 128 to the Transmit Packet Length register
        //fill the payload into the transmit FIFO
        //write 64bytes into the FIFO
        for(temp8=0;temp8<64;temp8++)
        {
                SpiWriteRegister(0x7F,tx_packet[temp8]);
        }

        //Disable all other interrupts and enable the FIFO almost empty interrupt only.
        SpiWriteRegister(0x05, 0x20);       //write 0x20 to the Interrupt Enable 1 register
        SpiWriteRegister(0x06, 0x00);       //write 0x00 to the Interrupt Enable 2 register

        //Read interrupt status regsiters. It clear all pending interrupts and the nIRQ pin goes back to high.
        ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
```

```
ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

/*enable transmitter*/
//The radio forms the packet and send it automatically.
SpiWriteRegister(0x07, 0x09);       //write 0x09 to the Operating Function Control 1 register

/*wait for TX FIFO almost empty interrupt*/
while(NIRQ == 1);

//TX FIFO almost empty interrupt occured -->
//write the next 32bytes into the FIFO
for(temp8=0;temp8<32;temp8++)
{
        SpiWriteRegister(0x7F,tx_packet[64+temp8]);
}
//Read interrupt status regsiters. It clear all pending interrupts and the nIRQ pin goes back to high.
ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register

/*Wait for TX FIFO almost empty interrupt*/
while(NIRQ == 1);

//TX FIFO almost empty interrupt occured -->
//write the last 32bytes into the FIFO
for(temp8=0;temp8<32;temp8++)               //copy the remain 32 byte to the FIFO
{
        SpiWriteRegister(0x7F,tx_packet[96+temp8]);
}

//Disable all other interrupts and enable the packet sent interrupt only.
//This will be used for indicating the successfull packet transmission for the MCU
SpiWriteRegister(0x05, 0x04);//write 0x04 to the Interrupt Enable 1 register
//Read interrupt status regsiters. It clear all pending interrupts and the nIRQ pin goes back to high.
ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register

/*wait for the packet sent interrupt*/
//The MCU just needs to wait for the 'ipksent' interrupt.
while(NIRQ == 1);
//Read interrupt status registers. It clear all pending interrupts and the nIRQ pin goes back to high.
ItStatus1 = SpiReadRegister(0x03);          //read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);          //read the Interrupt Status2 register

//wait a bit for showing the LED a bit longer
for(delay = 0; delay < 10000;delay++);
//turn off the LED
LED1 = 0;

//after packet transmission set the interrupt enable bits according receiving mode
//Enable two interrupts:
// a) one which shows that a valid packet received: 'ipkval'
// b) second shows if the packet received with incorrect CRC: 'icrcerror'
// c) third shows if the RX fifo almost full: 'irxffafull
SpiWriteRegister(0x05, 0x13);       //write 0x13 to the Interrupt Enable 1 register
SpiWriteRegister(0x06, 0x00);       //write 0x00 to the Interrupt Enable 2 register
//Read interrupt status registers. It clear all pending interrupts and the nIRQ pin goes back to high.
ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register
//set the Frequency Deviation register according to the AFC limiter
SpiWriteRegister(0x72, 0x1F);       //write 0x1F to the Frequency Deviation register

/*enable receiver chain again*/
SpiWriteRegister(0x07, 0x05);       //write 0x05 to the Operating Function Control 1 register}
```

6.3.3. Packet Reception

If an interrupt occurs, the MCU checks for the cause of the interrupt:

- In the case of a CRC error, the demo resets the receive FIFO, blinks the LEDs to show the error, and goes back into continuous receive mode.
- In the case of a FIFO Almost Full interrupt (“irxffafull” bit in the Interrupt Status 1 register), it means that the packet is being received and the MCU needs to read data from the receive FIFO. For each interrupt, the MCU reads 32 bytes from the receive FIFO to create space for the remaining bytes of the packet.
- If a packet received interrupt occurs, the MCU switches out of receive mode, checks whether the expected packet arrived, and blinks the RX LED. The demo then goes back into continuous receive mode.

```
//wait for the interrupt event
        //If it occurs, then it means a packet received or CRC error happened
        if( NIRQ == 0 )
        {
                //read interrupt status registers
                ItStatus1 = SpiReadRegister(0x03);//read the Interrupt Status1 register
                ItStatus2 = SpiReadRegister(0x04);//read the Interrupt Status2 register

                /*CRC Error interrupt occurred*/
                if( (ItStatus1 & 0x01) == 0x01 )
                {
                        //reset the RX FIFO
                        SpiWriteRegister(0x08, 0x02);
//write 0x02 to the Operating Function Control 2 register
                        SpiWriteRegister(0x08, 0x00);//write 0x00 to the Operating Function Control 2 register
                        //blink all LEDs to show the error
                        LED1 = 1;
                        LED2 = 1;
                        LED3 = 1;
                        LED4 = 1;
                        for(delay = 0; delay < 10000;delay++);
                        LED1 = 0;
                        LED2 = 0;
                        LED3 = 0;
                        LED4 = 0;
                }

                /*RX FIFO almost full interrupt occurred*/
                if( (ItStatus1 & 0x10) == 0x10 )
                {
                        //read 32bytes from the FIFO
                        for(temp8=pointer;temp8<pointer+32;temp8++)
                        {
                                rx_packet[temp8] = SpiReadRegister(0x7F);
                        }
                        //update receive buffer pointer
                        pointer = pointer + 32;
                }

                /*packet received interrupt occurred*/
                if( (ItStatus1 & 0x02) == 0x02 )
                {
                        //disable the receiver chain
                        SpiWriteRegister(0x07, 0x01);//write 0x01 to the Operating Function Control 1 register

                        //Read the length of the received payload
                        length = SpiReadRegister(0x4B);    //read the Received Packet Length register
```

```
                    //get the remaining 32bytes from the RX FIFO
                    for(temp8=pointer;temp8<pointer+32;temp8++)
                    {
                            rx_packet[temp8] = SpiReadRegister(0x7F);
                    }

                    //clear receive buffer pointer
                    pointer = 0;

                    //check whether the content of the packet is valid
                    if(( length == 128 ) && ( rx_packet[127] == 127 ))
                    {
                            //turn on the LED
                            LED1 = 1;
                            //wait a bit for showing the LED a bit longer
                            for(delay = 0; delay < 10000;delay++);
                            //turn off the LED
                            LED1 = 0;
                    }


                    /*enable receiver chain again*/
                    SpiWriteRegister(0x07, 0x05);
//write 0x05 to the Operating Function Control 1 register
                            }
                }
```

Note: The receive function for Si4431 is slightly different. See "4.2.1. Receiving a Packet with the Si4431 Revision A0" on page 29 for more details. The sample code for Si443x-B1 is slightly different since it has separate Frequency Deviation and AFC Limit Registers; so, there is no need to set them every time.

DOCUMENT CHANGE LIST

Revision 0.6 to Revision 0.7

- EZRadioPRO version B1 support added.

Simplicity Studio

One-click access to MCU tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

www.silabs.com/simplicity

MCU Portfolio
www.silabs.com/mcu

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com