AN633

Si446x Programming Guide and Sample Codes

1. Introduction

This document provides an overview of configuring the Si446x for transmitter, receiver, and transceiver operation using simple software example codes.

The following examples are covered in this programming guide:

- How to transmit a packet in simple Packet Handler mode (Sample Code 1)
- How to receive a packet in simple Packet Handler mode (Sample Code 1)
- How to transmit a packet in Packet Handler mode using a FIELD (Sample Code 2)
- How to receive a packet in Packet Handler mode using a FIELD (Sample Code 2)
- How to implement bidirectional packet-based communication (Sample Code 3)
- How to use variable packet length (Sample Code 3)
- How to use 4(G)FSK modulation (Sample Code 4)

Please contact the Wireless Support Team for the latest example source codes.

2. Hardware Options

The source code is provided for the Si4460-B0, Si4461-B0, Si4463-B0, and Si4464-B0 devices.

A separate Silicon Labs IDE* workspace is provided for each example on the UDP platform.

***Note:** Use IDE 4.40 or higher.

2.1. Test Card Options

The power amplifier and the LNA are not connected together inside the Si446x devices. Several different test cards are introduced to provide examples for main connection schemes.

On test cards using direct-tie connection, the TX and RX pins are directly connected externally, eliminating the need for an RF switch.

On test cards using an RF switch, separate transmit and receive pins on the RFIC are connected to an antenna via an SPDT RF switch for single antenna operation. The radio device assists with control of the RF switch. By routing the RX State and TX State signals to any two GPIOs, the radio can automatically control the RF switch. The GPIOs will disable the RF switch if the radio is not in active mode.

On test cards using split connection, the TX and RX pins are routed to different antenna connectors.

Test cards with external PA (FEM or FET type) are also provided by Silabs.

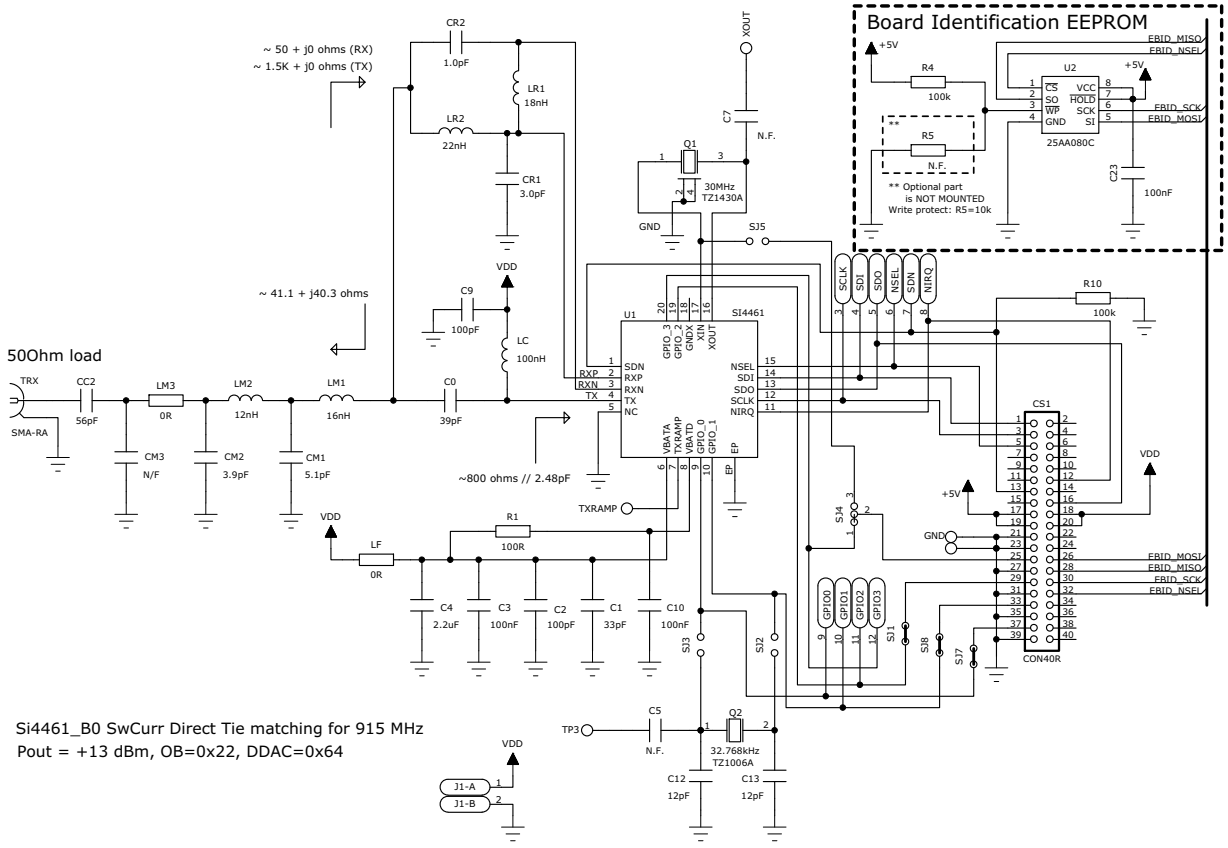


Figure 1. Direct Tie Si4461 Test Card Schematic

The example source code can be compiled for various test cards and radios. The following compiling option (located at the beginning of the *ezrp_next_api.h* file) selects the proper configuration (*#define EZRP_NEXT_TestCard TestCard_4463_TCE20C915*). The selection has to be made based on the marketing number of the test card.

- TestCard_4463_TSQ20D169
- TestCard_4463_TSQ27F169
- TestCard_4461_TCE14D434
- TestCard_4460_TCE10D434
- TestCard_4463_TCE20B460
- TestCard_4463_TCE20C460
- TestCard_4460_TSC10D868
- TestCard_4460_TCE10D868
- TestCard_4461_TSC14D868
- TestCard_4461_TCE16D868
- TestCard_4463_TCE20B868
- TestCard_4463_TCE20C868
- TestCard_4463_TCE27F868
- TestCard_4463_TCE20B915
- TestCard_4463_TCE20C915

- TestCard_4463_TCE30E915
- TestCard_4464_TCE20B420

Note: The transmit output of Si446x devices must 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. UDP Platform

The example source codes run on the Universal Development Platform (UDP). The UDP consists of several boards. Both the MCU card and the test card are plugged into the UDP Motherboard (UP-BACKPLANE-01EK). It provides power supply and USB connectivity for the development system. The MCU card (UPMP-F960-EMIF-EK) has several peripherals for simple software development (e.g. LEDs, Push Buttons, etc.), and it has a socket for various MCU selections. UPPI-F960-EK MCU Pico board is used for software development, which has a Si8051F960 MCU on it. It controls the radio on the test card using the SPI bus. The test card is connected to the 40-pin connector (J3) on the UDP Motherboard where the following signals are used (UDP_MB refers to UDP Motherboard, MCU refers to the MCU of the UPMP-F960-EMIF card):

Table 1. Si446x Pin Assignments

Si446x			UDP Platform
Pin Number	**Pin Name**	**Pin Function**	**Pin Name**
Exposed pad, 18	GND	Ground	Ground
6, 8	VDD	Supply input	PWR_RADIO through UDP_MB –J21
11	NIRQ	Interrupt output, active low	MCU / P1.6
1	SDN	Shutdown input, active high	MCU / P1.7
15	NSEL	SPI select input	MCU / P2.3
12	SCLK	SPI clock input	MCU / P2.0
14	SDI	SPI data input	MCU / P2.2
13	SDO	SPI data output	MCU / P2.1

For more info on the UDP platform, refer to the “UDP Motherboard User's Guide”.

2.2.1. Setting up the Development Board

Before downloading example code into the MCU, the UDP platform must be configured according to the following instructions:

1. Connect the Si446x test card to the 40-pin (J3) connector.
2. Connect the antenna(s) to the SMA connector(s) of the test card.
3. Set S5 to OFF to disable the 6.5 V voltage on pin 36 of the test card connector.
4. Set SW5 to UDP on the MCU card.
5. Set SW7 and SW12 to VBAT on the MCU card.
6. Short JP19 and JP21 jumpers as shown in Figure 2 to provide VDD for the radio (the current consumption of the radio can be measured through these jumpers).

Figure 2. Jumper Positions (JP19, JP21)

2.2.2. Downloading and Running the Example Codes

1. Connect the USB debug adapter to the 10-pin J13 connector of the MCU card.
2. To power the board, connect a 9 V dc power supply to J20 (UDP_MB), or connect the development board to a PC over USB cable (J16 on UDP_MB).
3. Turn on S3 on UDP_MB (Power switch).
4. Start Silicon Labs IDE on your computer.
5. Connect the IDE to the C8051F960 MCU of the development board by pressing the Connect button on the toolbar or by invoking the menu Debug → Connect menu item.
6. Erase the FLASH of the C8051F960 MCU in the Debug → Download object code → Erase all code space menu item.
7. Download the desired example HEX file either by hitting the Download code (Alt+D) toolbar button or from the Debug → Download object code menu item.
8. Hit the Disconnect toolbar button or invoke the Debug → Disconnect menu item to release the device from halt and let it run.

Figure 3. J13 Connector on the MCU Card

2.3. Using the Resources of the Development Board

The example programs are written in “C”. General headers and code files are provided to ease the programming of the development board.

The file called *compiler_defs.h* contains macro definitions to accommodate 8051 compiler differences and declaring special function registers and other 8051-specific features. Using these macros allows the example code to be compiled with several common C compilers, such as Keil, SDCC, Raisonance, etc.

2.3.1. Initialization

The file called *C8051F960_defs.h* contains the register/bit definitions for the C8051F96x MCU family.

The file called *hardware_defs.h* contains the UDP platform-specific definitions.

The following macros in the *hardware_defs.h* file assign a label to the given IO port, so they can be more easily referenced in the source code. For example, the *SBIT(EZRP_PWRDN, SFR_P1, 7);* line assigns the EZRP_PWRDN label to P1.7 (refer to Table 1), which the SDN pin of the radio is connected to:

```
SBIT(MCU_SDA, SFR_P0, 6);
SBIT(MCU_SCL, SFR_P0, 7);

SBIT(MCU_SCK, SFR_P2, 0);
SBIT(MCU_MISO,SFR_P2, 1);
SBIT(MCU_MOSI,SFR_P2, 2);
SBIT(MCU_NSEL, SFR_P2, 3);

SBIT(EZRP_NIRQ, SFR_P1, 6);
SBIT(EZRP_PWRDN, SFR_P1, 7);

SBIT(PB1, SFR_P3, 0);
SBIT(PB2, SFR_P3, 1);
SBIT(PB3, SFR_P3, 2);
SBIT(PB4, SFR_P3, 3);
SBIT(LED1, SFR_P3, 0);
SBIT(LED2, SFR_P3, 1);
SBIT(LED3, SFR_P3, 2);
SBIT(LED4, SFR_P3, 3);
```

The *void MCU_Init(void)* function in the *main.c* file initializes all the peripherals necessary for the example programs.

UDP platform provides an own SPI interface for the test card connected to J3 connector, other SPI devices should not conflict with the test card. SPI interface connected to the radio test card are referred as SPI “0” in the sample codes.

2.3.2. LED and Push Buttons

The LED control and push button reading functions can be found in the *control_IO.c* file:

- *void SetLed(U8 number)*
- *void ClearLed(U8 number)*
- *bit GetPB(U8 number)*

Reading the push buttons is not trivial, as a button and an LED share the same pin of the MCU. During the short period of reading a push button, the pin is set high and its direction is changed to input. This change may cause a brief but unnoticeable LED state change. After the push button is checked, the original LED status is restored.

3. Using the Si446x Radios

This chapter describes the details of how to operate the Si446x radios. As mentioned above, there are four software example codes provided; these are referred to as Sample Codes 1–4 later in this document. Many features are common in all of these sample codes such as controlling the radio (Control the Radio), using the application programming interface (API), initializing the MCU, while others are only used in certain examples, like using the Packet Handler mode, or the so called FIELDs.

3.1. Controlling the Radio

The Si446x radios have several pins to interface the host MCU.

Four pins, SDI, SDO, SCLK, NSEL, are used to control the radio over the SPI bus. The user has access to an Application Programming Interface (API) via the SPI bus, which is described in detail in the Application Programming Interface section below.

The SDN (Shutdown) pin is used to completely disable the radio (when the pin is pulled high) and put the device into the lowest power consumption state. After the SDN pin is pulled low the radio wakes up and performs a Power On Reset. It takes a maximum of 5 ms until the chip is ready to receive commands on the SPI bus (GPIO1 pin goes high when the radio is ready for receiving SPI commands). When SDN is high and the radio is in shutdown state, the GPIOs are set to drive an output low level.

The radio has an interrupt output pin, NIRQ, which can be used to promptly notify the host MCU of multiple events. The NIRQ pin is active low, and goes back to high if the pending interrupt flag is cleared by reading the appropriate Interrupt Pending registers.

3.2. Application Programming Interface

The Si446x radios can be controlled over the SPI interface. For efficient communication, the API (Application Programming Interface) is implemented to interact with the radio. To invoke the API, a power-up command must be sent to the radio after the POR is complete (it takes TPower_on = 5 ms + 6..10 ms to complete the entire power on sequence (POR + invoke API)).

The programming interface allows the user to do the following:

- Send commands to the radio.
- Read status information.
- Set and get radio parameters.
- Handle the transmit and receive FIFOs.

The following two sections describe the SPI transactions of sending commands and getting information from the chip.

3.2.1. Sending Commands

The behavior of the radio can be changed by sending commands to the radio (e.g. changing the power states, start packet transmission, etc.). The radio can be configured through several so called “properties”. The properties hold radio configuration settings, such as interrupt settings, modem parameters, packet handler settings, etc. The properties can be set and read via API commands.

Most of the commands are sent from the host MCU to the radio, and the host MCU does not expect any response from the chip. There are other commands that are used to read back a property from the chip, such as checking the interrupt status flags, reading the transmit/receive FIFOs, etc. This paragraph gives a detailed overview about the simple commands (commands with no response); the Get response to a command paragraph describes the SPI transactions of getting information from the radio (commands with response).

All the commands that are sent to the radio should have the same structure. After pulling down the NSEL pin of the radio, the command ID should be sent first. The commands may have up to 15 input parameters with the exception of *START_RX* and *START_TX*, which may have 64 bytes input. The number of input parameters varies depending on the actual command. The NSEL pin must be pulled high when the transaction finishes.

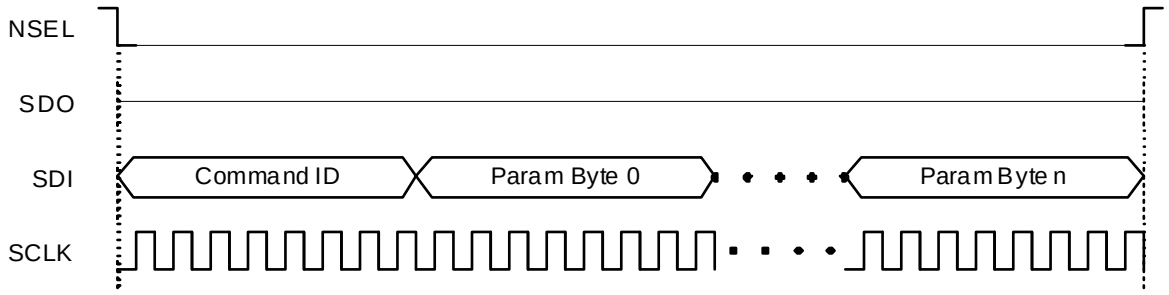


Figure 4. Send Command SPI Transaction

The command IDs are listed in Table 2.

Table 2. Commands

Command ID	Name	Input Parameters
0x02	POWER_UP	Power-up device and mode selection. Modes include operational function.
0x04	PATCH_IMAGE	Loads image from NVM/ROM into RAM
0x00	NOP	No operation command
0x01	PART_INFO	Reports basic information about the device
0x10	FUNC_INFO	Returns the Function revision information of the device
0x11	SET_PROPERTY	Sets the value of a property
0x12	GET_PROPERTY	Retrieve a property's value
0x13	GPIO_PIN_CFG	Configures the GPIO pins
0x15	FIFO_INFO	Provides access to transmit and receive FIFO counts and reset
0x17	IRCAL	Calibrate IR
0x20	GET_INT_STATUS	Returns the interrupt status byte
0x33	REQUEST_DEVICE_STATE	Request current device state
0x34	CHANGE_STATE	Update state machine entries
0x14	GET_ADC_READING	Retrieve the results of possible ADC conversions
0x16	GET_PACKET_INFO	Returns information about the last packet received
0x18	PROTOCOL_CFG	Sets the chip up for specified protocol
0x21	GET_PH_STATUS	Returns the packet handler status
0x22	GET_MODEM_STATUS	Returns the modem status byte
0x23	GET_CHIP_STATUS	Returns the chip status
0x36	RX_HOP	Fast RX to RX transitions for use in frequency hopping systems

Table 2. Commands (Continued)

Command ID	Name	Input Parameters
0x50	FRR_A	Returns Fast response register A
0x51	FRR_B	Returns Fast response register B
0x53	FRR_C	Returns Fast response register C
0x57	FRR_D	Returns Fast response register D
0x44	READ Command buffer	Returns Clear to send (CTS) value and the result of the previous command
0x66	TX_FIFO_WRITE	Writes TX data buffer (max. 64 bytes)
0x77	RX_FIFO_READ	Reads RX data buffer (max. 64 bytes)
0x31	START_TX	Switches to TX state and starts packet transmission.
0x32	START_RX	Switches to RX state.

After the radio receives a command, it processes the request. During this time, the radio is not capable of receiving a new command. The host MCU has to poll the radio and identify when the next command can be sent. The CTS (Clear to Send) signal shows the actual status of the command buffer of the radio. It can be monitored over the SPI port or on GPIOs, or the chip may generate an interrupt if it is ready to receive the next command. These three options are detailed below.

3.2.1.1. CTS Ready via SPI

In order to make sure that the radio is ready to receive the next command, the host MCU has to pull down the NSEL pin to monitor the status of CTS over the SPI port. The 0x44 command ID has to be sent, and eight clock pulses have to be generated on the SCLK pin. During the additional eight clock cycles, the radio clocks out the CTS as a byte on the SDO pin. When complete, the NSEL should be pulled back to high. If the CTS byte is 0xFF, it means that the radio processed the command successfully and is ready to receive the next command; in any other case, the CTS read procedure has to be repeated from the beginning as long as the CTS byte is not 0xFF. Note that commands without any input parameters require a dummy byte for proper CTS responses.

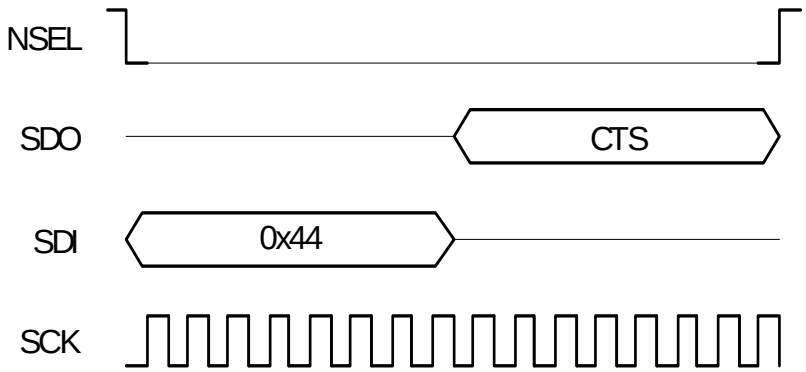


Figure 5. CTS Read over SPI after a Command is Sent

3.2.1.2. CTS Ready via GPIOs

Any of the GPIOs can be configured for monitoring the CTS. GPIOs can be configured to go high or low if the chip completes the command. The function of the GPIOs can be changed by the *GPIO_PIN_CFG* command. By default, GPIO1 is set as “High when command completed, low otherwise” after Power On Reset. Therefore, this pin can be used for monitoring the CTS right after Power On Reset to know when the chip is ready to boot up.

3.2.1.3. CTS Ready via Interrupts

The radio asserts the *CHIP_READY* interrupt flag if a command is completed. The interrupt flag can be monitored by either the *GET_CHIP_STATUS*, or the *GET_INT_STATUS* command. In addition to this, the *CHIP_STATUS* and *CHIP_INT_STATUS_PEND* bytes can be assigned to one of the Fast Response Registers which provides faster access to them (more details can be found about the Fast Response registers in "3.2.4. Using the Fast Response Registers" on page 11). Apart from monitoring the interrupt flags, the radio may pull down the NIRQ pin if this feature is enabled.

If a new command is sent while the CTS is asserted, then the radio ignores the new command. The Si446x can generate an interrupt about this to communicate the error to the MCU: CMD_ERROR interrupt flag in the *CHIP_STATUS* group. The interrupt flag has to be read (by issuing a *GET_CHIP_STATUS* or *GET_INTERRUPT_STATUS* command) to clear the pending interrupt and release the NIRQ pin. No other action is needed to reset the command buffer of the radio, but, after a *CMD_ERROR*, the host MCU should repeat the new command after the radio has processed the previous one.

3.2.2. Get Response to a Command

Reading from the radio requires several steps to be followed. The host MCU should send a command with the address it requests to read. The radio holds the CTS while it retrieves the requested information. Once the CTS is set (0xFF), the host MCU can read the answer from the radio.

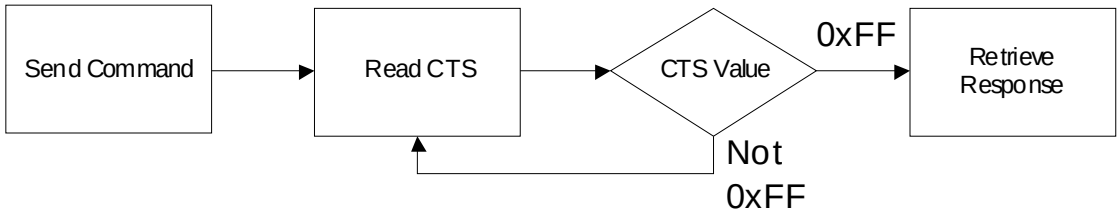


Figure 6. Read Procedure

After sending the command, the NSEL pin has to be pulled high, as described in "3.2.1. Sending Commands" on page 6. There are several ways to decide if the Si446x is ready with the response.

If the CTS is polled on the GPIOs, or the radio is configured to provide interrupt if the answer is available; then, the response can be read out from the radio with the following SPI transaction:

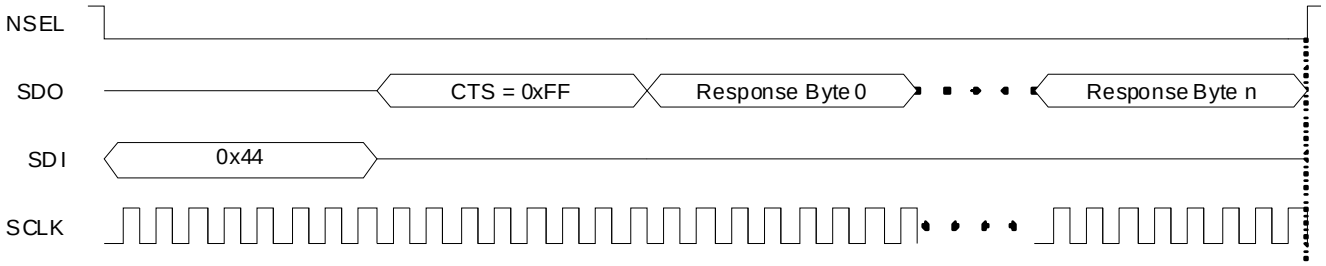


Figure 7. Read the Response from the Radio

If the CTS is polled over the SPI bus, the host MCU should pull the NSEL pin low. This should be followed by sending out the 0x44 Read command ID and providing an additional eight clock pulses on the SCLK pin. The radio will provide the CTS byte on its SDO pin during the additional clock pulses. If the CTS byte is 0x00, then the response is not yet ready and the host MCU should pull up the NSEL pin and repeat the procedure from the beginning as long as the CTS becomes 0xFF. If CTS is 0xFF, then the host MCU should keep the NSEL pin low and provide as many clock cycles on the SCLK pin as the data to be read out requires. The radio will clock out the requested data on its SDO pin during the additional clock pulses.

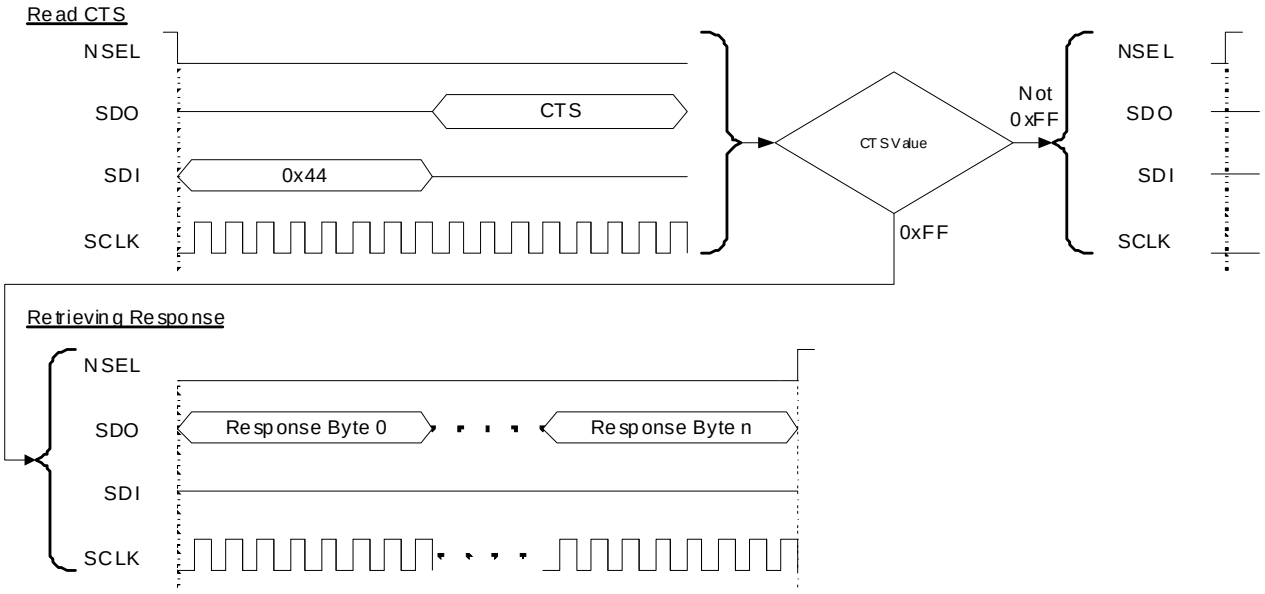


Figure 8. Monitor CTS and Read the Response on the SPI Bus

Reading the response from the radio can be interrupted earlier. For example, if the host MCU asked for five bytes of response, it may read fewer bytes in one SPI transaction. As long as a new command is sent, the radio keeps the response for the last request in the command buffer. The host MCU can read the response several times in a new SPI transaction. In such a case, the response is always provided from the first byte.

Notes:

1. Up to 16 bytes of response can be read from the radio in one SPI transaction. If more bytes are read, the radio will provide the same 16 bytes of response in a circular manner.
2. If the command says that the host MCU expects N bytes of response, but, during the read sequence, the host MCU provides less than N bytes of clock pulses, it causes no issue for the radio. The response buffer is reset if a new command is issued.
3. If the command says that the host MCU expects N bytes of response, but, during the read sequence, the host MCU provides more than N bytes of clock pulses, the radio will provide unpredictable bytes after the first N bytes. The host MCU does not need to reset the SPI interface; it happens automatically if NSEL is pulled low before the next command is sent.

3.2.3. Write and Read the FIFOs

There are two 64-byte FIFOs for RX and TX data in the Si446x.

To fill data into the transmit FIFO, the host MCU should pull the NSEL pin low and send the 0x66 Transmit FIFO Write command ID followed by the bytes to be filled into the FIFO. Finally, the host MCU should pull the NSEL pin high. Up to 64 bytes can be filled into the FIFO during one SPI transaction.

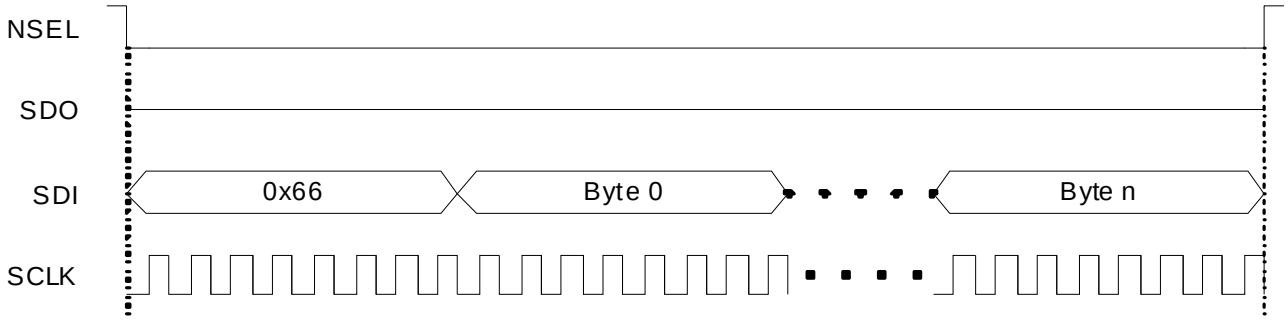


Figure 9. Transmit FIFO Write

If the host MCU wants to read the receive FIFO, it has to pull the NSEL pin low and send the 0x77 Receive FIFO Read command ID. The MCU should provide as many clock pulses on the SCLK pin as necessary for the radio to clock out the requested amount of bytes from the FIFO on the SDO pin. Finally, the host MCU should pull up the NSEL pin.

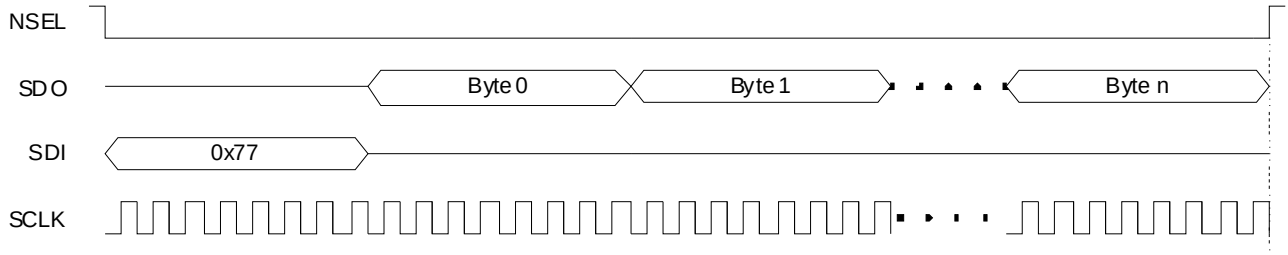


Figure 10. Receive FIFO Read

If more than 64 bytes are written into the Transmit FIFO, then a FIFO overflow occurs. If more bytes are read from the Receive FIFO than it holds, then FIFO underflow occurs. In either of these cases, the *FIFO_UNDERFLOW_OVERFLOW_ERROR* interrupt flag will be set. The radio can also generate an interrupt on the NIRQ pin if this flag is enabled. The interrupt flag has to be read, by issuing a *GET_CHIP_STATUS* or *GET_INTERRUPT_STATUS* command, to clear the pending interrupt and release the NIRQ pin.

3.2.4. Using the Fast Response Registers

There are several types of status information that can be read out from the radio faster than the method defined in "3.2.2. Get Response to a Command" on page 9. Four Fast Response Registers are available for this purpose. The *FRR_CTL_x_MODE* (where x can be A, B, C or D) properties define what status information is assigned to a given Fast Response Register (FRR).

The actual value of the registers can be read by pulling down the NSEL pin, issuing the proper command ID, and providing an additional eight clock pulses on the SCLK pin. During these clock pulses, the radio provides the value of the addressed FRR. The NSEL pin has to be pulled high after finishing the register read.

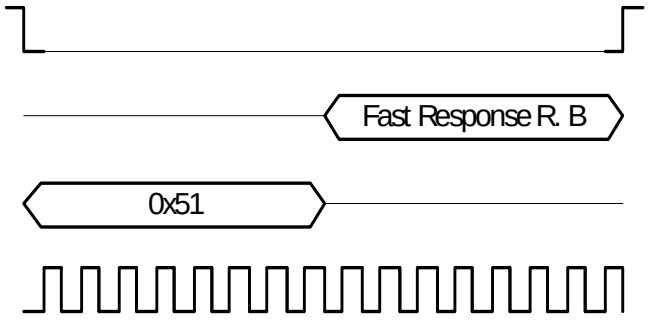


Figure 11. Reading a Single Fast Response Register

It is also possible to read out multiple FRRs in a single SPI transaction. The NSEL pin has to be pulled low, and one of the FRRs has to be addressed with the proper command ID. Providing an additional 8 x N clock cycles will clock out an additional N number of FRRs. After the fourth byte is read, the radio will provide the value of the registers in a circular manner. The reading stops by pulling the NSEL pin high.

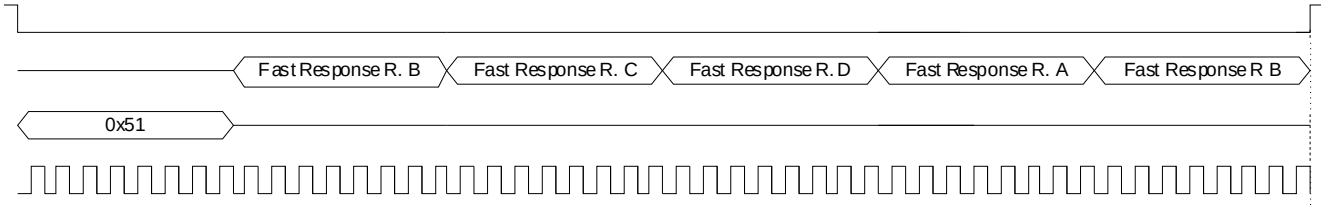


Figure 12. Reading More Fast Response Registers in a Single SPI Transaction

Note: If the pending interrupt status register is read through the FRR, the NIRQ pin does not go back to high. The pending interrupt registers have to be read by a Get response to a command sequence in order to release the NIRQ pin.

3.2.5. Using the API in the Sample Codes

The API functions and definitions can be found in *ezrp_next_api.c* and *ezrp_next_api.h*.

Transmitting a command can be initiated by the *U8 bApi_SendCommand(U8 bCmdLength, U8 *pbCmdData)* function. The actual command to be sent to the chip has to be stored in a byte array. The *pbCmdData* pointer is an input parameter, which points to the beginning of the array. The number of bytes sent as a command has to be defined by the *bCmdLength* input parameter.

```
U8 bApi_SendCommand(U8 bCmdLength, U8 *pbCmdData)        // Send a command + data to the chip
{
        SpiClearNsel(0);                                 // select radio IC by pulling its nSEL pin low
        bSpi_SendDataNoResp(bCmdLength, pbCmdData);      // Send data array to the radio IC via SPI
        SpiSetNsel(1);                                   // de-select radio IC by putting its nSEL pin high
        return 0;
}
```

The *U8 vApi_WaitforCTS(void)* function polls the status of the CTS over the SPI bus. It is used to decide whether the radio processed the command. If the radio processed the command, the function returns with 0x00. Otherwise, it checks the CTS up to 2500 times and returns with 0x01. It can be considered an error, and the radio needs to be reset.

```
U8 vApi_WaitforCTS(void)
{
        SEGMENT_VARIABLE(bCtsValue, U8, SEG_XDATA);
        SEGMENT_VARIABLE(bErrCnt, U16, SEG_XDATA);

        bCtsValue = 0;
        bErrCnt = 0;

        while (bCtsValue!=0xFF)                          // Wait until radio IC is ready with the data
        {
                SpiClearNsel(0);                         // select radio IC by pulling its nSEL pin low
                bSpi_SendDataByte(0x44);                 // Read command buffer; send command byte
                bSpi_SendDataGetResp(1, &bCtsValue);     // Read command buffer; get CTS value
                SpiSetNsel(1);                           // If CTS is not 0xFF, put NSS high and stay in waiting
                if (++bErrCnt > MAX_CTS_RETRY)
                {
                        return 1;                        // Error handling; if wrong CTS reads exceeds a limit
                }
        }
        return 0;
}
```

The *U8 bApi_GetResponse(U8 bRespLength, U8 *pbRespData)* function waits for the chip to process the read command by polling the CTS over the SPI bus. If the response is available, the function keeps the NSEL pin low and reads *bRespLength* number of bytes from the radio. The result bytes are stored in the memory location defined by the *pbRespdata* pointer, and the function returns with 0x00. The function returns with 0x01 if the radio is not ready with the response after 2500 attempts. It can be considered an error, and the radio needs to be reset.

```
U8 bApi_GetResponse(U8 bRespLength, U8 *pbRespData)      //Get response from the chip (used after a command)
{
        SEGMENT_VARIABLE(bCtsValue, U8, SEG_XDATA);
        SEGMENT_VARIABLE(bErrCnt, U16, SEG_XDATA);

        bCtsValue = 0;
        bErrCnt = 0;

        while (bCtsValue!=0xFF)                 // Wait until radio IC is ready with the data
        {
                SpiClearNsel(0);                // select radio IC by pulling its nSEL pin low
                bSpi_SendDataByte(0x44);        // Read command buffer; send command byte
                bSpi_SendDataGetResp(1, &bCtsValue);
                if(bCtsValue != 0xFF)
                {
                        SpiSetNsel(1);
                }
        }
        if(bErrCnt++ > MAX_CTS_RETRY)
        {
                return 1;
        }
        bSpi_SendDataGetResp(bRespLength, pbRespData); // CTS value ok, get the response data from the radio IC
        SpiSetNsel(1);                                  // de-select radio IC by putting its nSEL pin high

        return 0;
}
```

The Transmit FIFO is filled by the *U8 bApi_WriteTxDataBuffer(U8 bTxFifoLength, U8 *pbTxFifoData)* function. A byte array has to be created and the data needs to be stored in that prior to calling the function. The number of bytes to be filled into the FIFO is defined by the *bTxFifoLength* input parameter, while the *pbTxFifoData* pointer has to point to the beginning of the byte array.

```
U8 bApi_WriteTxDataBuffer(U8 bTxFifoLength, U8 *pbTxFifoData)          // Write Tx FIFO
{
        SpiClearNsel(0);                                        // select radio IC by pulling its nSEL pin low
        bSpi_SendDataByte(0x66);                                // Send Tx write command
        bSpi_SendDataNoResp(bTxFifoLength, pbTxFifoData);       // Write Tx FIFO
        SpiSetNsel(1);                                          // de-select radio IC by putting its nSEL pin high

        return 0;
}
```

The Receive FIFO can be accessed by the *U8 bApi_ReadRxDataBuffer(U8 bRxFifoLength, U8 *pbRxFifoData)* function. The *bRxFifoLength* input parameter defines how many bytes the function needs to read from the FIFO. The result is stored in the memory location defined by the *pbRxFifoData* pointer.

```
U8 bApi_ReadRxDataBuffer(U8 bRxFifoLength, U8 *pbRxFifoData)
{
        SpiClearNsel(0);                                    // select radio IC by pulling its nSEL pin low
        bSpi_SendDataByte(0x77);                            // Send Rx read command
        bSpi_SendDataGetResp(bRxFifoLength, pbRxFifoData);  // Write Tx FIFO
        SpiSetNsel(1);                                      // de-select radio IC by putting its nSEL pin high

        return 0;
}
```

The *U8 bApi_GetFastResponseRegister(U8 bStartReg, U8 bNmbrOfRegs, U8 * pbRegValues)* function reads the Fast Response Registers. The *bStartReg* input parameter defines the first register to be read. The name of the Fast Response Registers is defined in the *Si446x_B0_defs.h* header file:

```
#define CMD_FAST_RESPONSE_REG_A     0x50
#define CMD_FAST_RESPONSE_REG_B     0x51
#define CMD_FAST_RESPONSE_REG_C     0x53
#define CMD_FAST_RESPONSE_REG_D     0x57
```

The number of register to be read is defined by the *bNmbrOfRegs* input parameter. The function stores the actual value of the FRR from the memory location defined by the *pbRegValues* pointer.

```
U8 bApi_GetFastResponseRegister(U8 bStartReg, U8 bNmbrOfRegs, U8 * pbRegValues)
{
        if((bNmbrOfRegs == 0) || (bNmbrOfRegs > 4))
        {
                return 1;
        }

        SpiClearNsel(0);
        bSpi_SendDataByte(bStartReg);
        bSpi_SendDataGetResp(bNmbrOfRegs, pbRegValues);
        SpiSetNsel(1);
        return 0;
}
```

3.3. Packet Transmitting Using the Packet Handler (Sample Code 1)

The software example uses push buttons on the hardware platform to initiate a packet transmission. After power on reset and invocation of the API, the firmware initializes the MCU and the radio IC. The radio is set to low power SLEEP mode. If any of the push buttons are pressed on the Motherboard, the host MCU wakes up the radio IC, sends a packet, then sends the radio IC back to low power SLEEP mode and returns to polling the push buttons.

The flowchart of the Sample Code 1 (TX side) is shown in Figure 13.

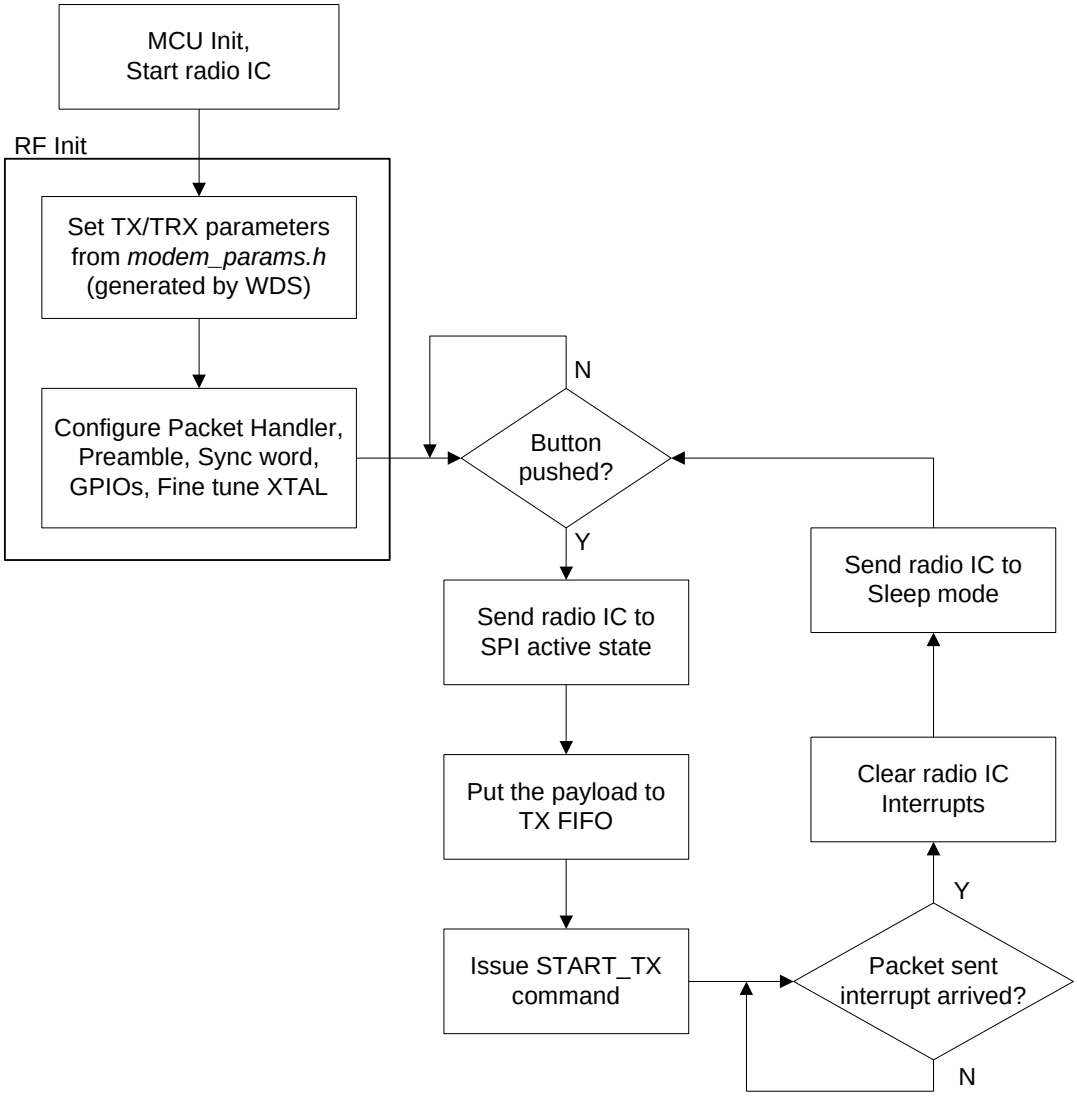


Figure 13. Transmitter Side Flowchart

The EZRadioPRO devices provide a flexible packet handler that supports a wide range of packet configuration options including the following:

- Programmable preamble length and pattern (up to 255 bytes)
- Programmable synchronization word length and pattern (up to four bytes)
- Automatic header generation and qualification (packet length, CRC)
- Fixed and variable length packets
- Packet fields to assign individual properties to different parts of the payload
- Payload data of up to 255 bytes (using the built-in Transmit, Receive FIFOs)
- Automatic CRC calculation and verification (different polynomials, up to 32 bits CRC)

In Sample Code 1, the packet handler uses fixed packet length with standard preamble and 2 bytes of synchronization word. The packet configuration used for the transmission is shown in Figure 14.

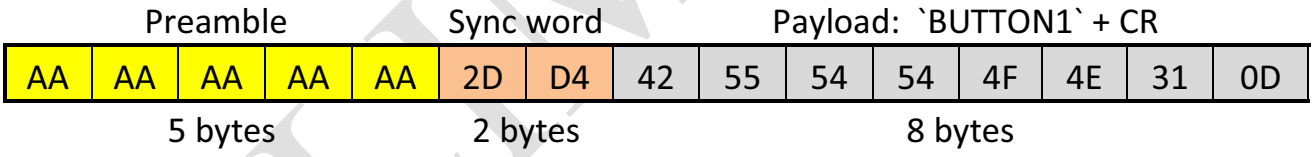


Figure 14. Packet Configuration

The payload of the packet depends on the push button that was pressed.

In order for the receiver to detect that a valid packet is present, the transmitted packet should start with a preamble and a synchronous pattern.

- The preamble is a continuous 1010 sequence; it is 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 can be settled correctly. The minimum required preamble length is application-dependent.
- After the transmitter and receiver have synchronized, the receiver has to determine where the payload data in the packet starts. The transmitter includes a known bit pattern to help identify the payload data; this is called the synchronous word.

The receiver also needs to know how long the transmitted packet is going to be. There are several options 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 this example code, fixed packet lengths are used; therefore, the length of the payload is not included in the packet.

3.3.1. Start the Radio

The EZRadioPRO radio has a built-in FIFO for packet transmission. Once the basic radio parameters are configured, only the payload data needs to be sent to the transmit FIFO of the radio; the packet transmission has to be started, and the radio will transmit the data and go back to the selected power state. The radio can optionally generate an interrupt when the packet transmission finishes.

This section describes how the EZRadioPRO can be configured to send the packet shown in Figure 14. Sample Code 1 example project is used to demonstrate packet transmission. The code segments used in this chapter are copied from the *main.c* file.

The radio IC has to be started or restarted in the following cases:

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

Figure 15 shows how to start the radio.

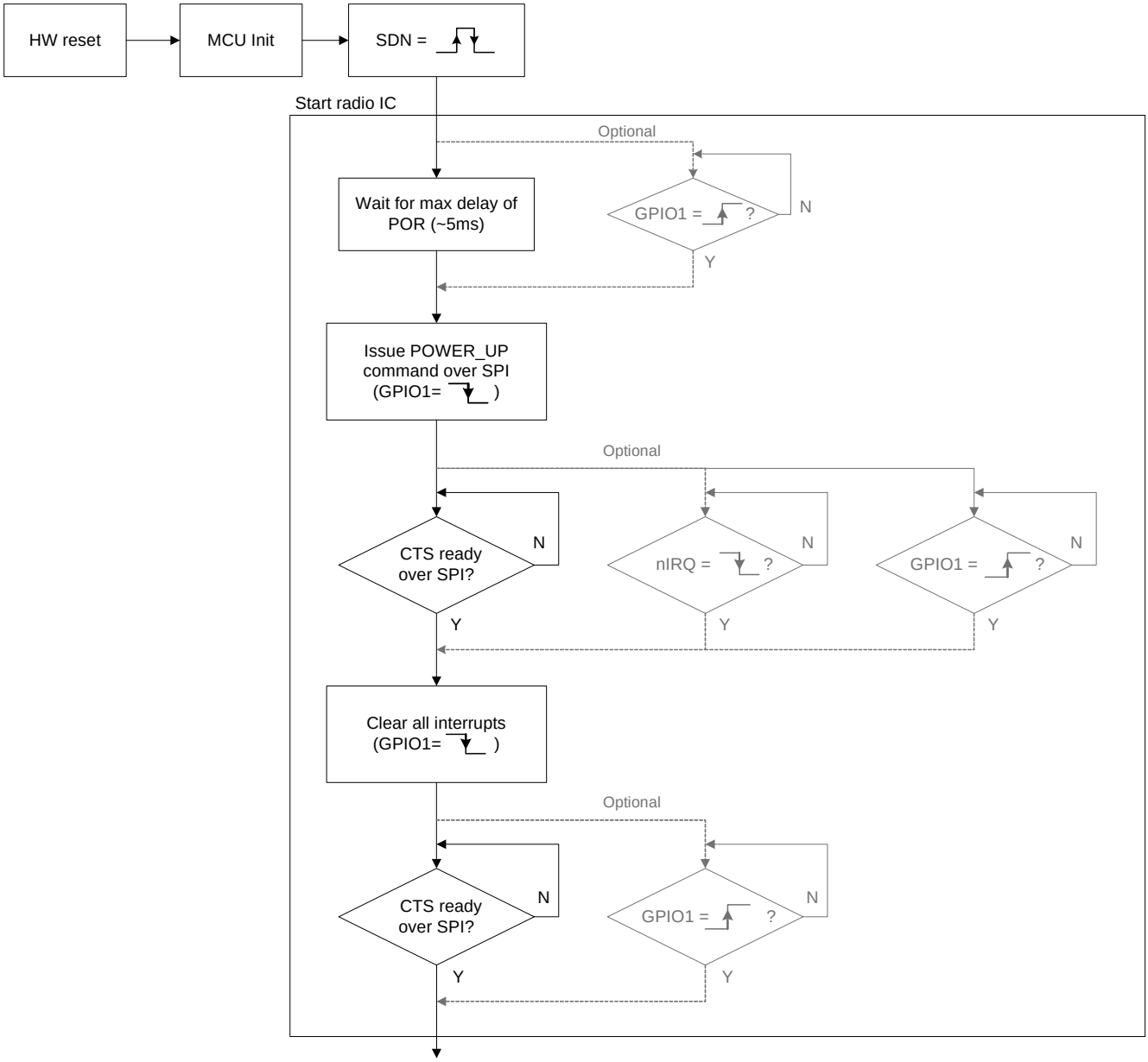


Figure 15. Start the Radio IC

The power-down reset cycle of the EZRadioPRO devices takes about 1 ms. During this reset period, the radio 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 host microcontroller to wait at least 1 ms.
- Connect the GPIO1 pin of the radio to the host MCU, and poll the status of this pin. During power on reset, it remains low. Once the reset is finished, the radio sets its state to high.

```
//Initialize the MCU:
//          - set IO ports for the Software Development board
//          - set MCU clock source
//          - initialize the SPI port
MCU_Init();

// Reset the radio
EZRP_SDN = 1;
// Wait ~300us (SDN pulse width)
for(wDelay=0; wDelay<330; wDelay++);
// Wake up the chip from SDN
EZRP_SDN = 0;

// Wait for POR (power on reset); ~5ms
for(wDelay=0; wDelay<5500; wDelay++);
```

Note: The IO pins status of the host MCU is not guaranteed before they got initialized in the MCU_Init() function. Therefore the power-down pin of the radio is set to high then low to perform a complete power on reset.

After a power on reset, the EZRadioPRO devices have to be sent to active mode by issuing a *POWER_UP* command via the SPI interface. It takes approximately 14 ms to complete the command (boot time), and can be monitored in three ways:

- GPIO1 pin of the radio goes low by issuing the command and the radio sets it to high if the command is completed
- The nIRQ pin is asserted if the command is completed
- The host MCU can monitor CTS over the SPI port to check when the command is completed:

```
// Start the radio
abApi_Write[0] = CMD_POWER_UP;                  // Use API command to power up the radio IC
abApi_Write[1] = 0x01;                          // Write global control registers
abApi_Write[2] = 0x00;                          // Write global control registers
bApi_SendCommand(3,abApi_Write);                // Send command to the radio IC
// Wait for boot
if (vApi_WaitforCTS())                          // Wait for CTS
 {
        while (1) {}    //Stop if PRO NG power-up error
 }
```

Note: CTS is polled after sending the *POWER_UP* API command. If CTS does not arrive within 2500 attempts, the program stops.

When the boot is complete, the radio asserts the *CHIP_READY_PEND* interrupt flag, and the NIRQ pin is set to low. The flag can be cleared and NIRQ released if the interrupt flag is read by the *GET_CHIP_STATUS* or *GET_INT_STATUS* commands.

```
// Read ITs, clear pending ones
abApi_Write[0] = CMD_GET_INT_STATUS;            // Use interrupt status command
abApi_Write[1] = 0;                             // Clear PH_CLR_PEND
abApi_Write[2] = 0;                             // Clear MODEM_CLR_PEND
abApi_Write[3] = 0;                             // Clear CHIP_CLR_PEND
bApi_SendCommand(4,abApi_Write);                // Send command to the radio IC
bApi_GetResponse(8,abApi_Read);                 // Make sure that CTS is ready, then get the response
```

3.3.2. Set RF Parameters

The following code sections set the basic RF parameters, such as center frequency, transmit data rate ,and transmit deviation.

There are a couple properties that must be set accurately for the radio to exhibit the desired RF behavior. These properties are in the MODEM and FREQUENCY property groups. Silicon Labs provides a PC tool (WDS - Si446x Radio Control Panel), which calculates the RF setting properties for Transmit and Receive operations. WDS can generate a header file for the proper property settings; it is called *modem_params.h*.

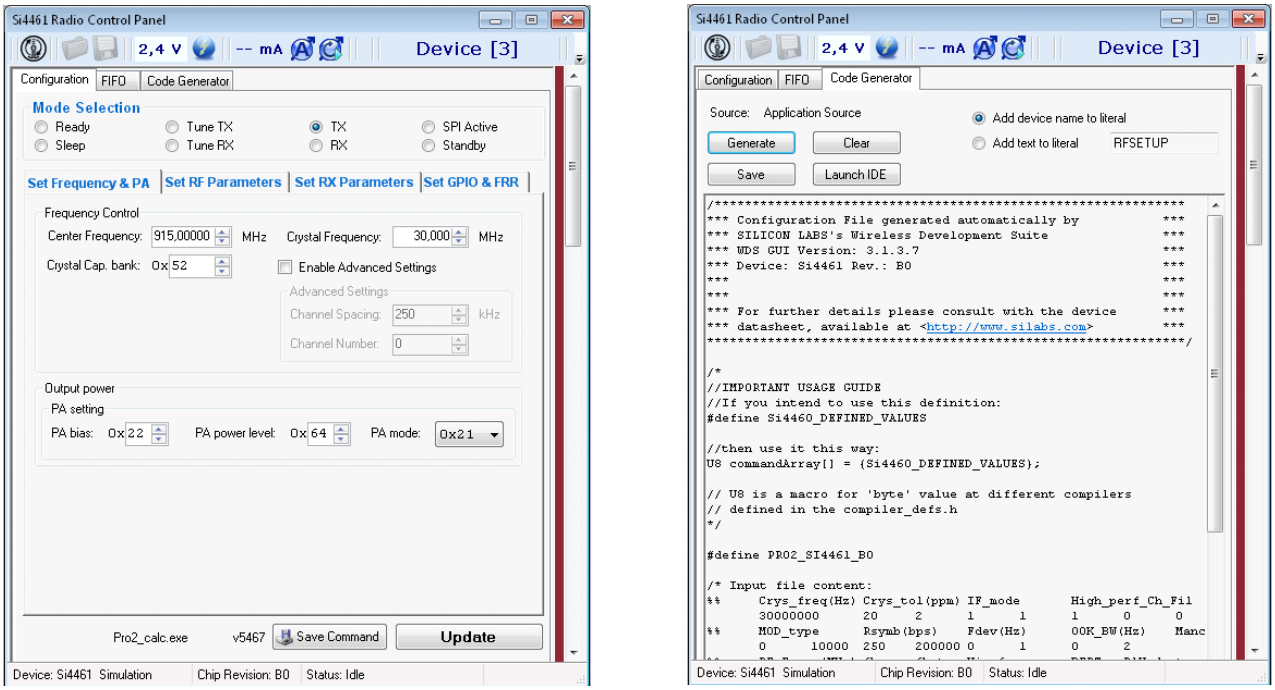


Figure 16. Generating modem_params.h header file in WDS

The modem_params.h header file contains definitions for the properties. Many properties are declared as definitions to minimize the SPI communication between the host MCU and the radio. The format of the definition is as follows:

- The name of the definition is the name of the first property of the given definition sets.
- The name of the definition ends with a number, which is the number of total bytes defined by the given definition.
- The first byte is the command ID of the *SET_PROPERTY* command.
- The following three bytes follows the *SET_PROPERTY* command requirements: the address of the first property and the number of properties set in the given command.
- Finally, the values of the properties set by the command.

FREQ_CONTROL_INTE property setting from modem_params.h: SET_PROPERTY command is 0x11. The property address is 0x4000; eight bytes are to be set, which are 0x1B, 0x0F...0xFF.

```
#define FREQ_CONTROL_INTE_12 0x11, 0x40, 0x08, 0x00,  0x1B, 0x0F, 0x8B, 0xF2, 0x00, 0x00, 0x20,  0xFF
```

Figure 17. Center Frequency Definition in the modem_params.h

To utilize the definitions, a byte array has to be defined in the C file. The array has to be filled up with the content of the given definition. The following code section shows how they are referenced at the beginning of the main.c example code:

```
// Set up modem parameters database; data is retrieved from modem_params.h header file which is
// automatically generated by the WDS (Wireless Development Suite)
code U8 ModemTrx1[] = {7, MODEM_MOD_TYPE_7};
code U8 ModemTrx2[] = {5, MODEM_CLKGEN_BAND_5};
code U8 ModemTrx3[] = {11, SYNTH_PFDCP_CPFF_11};
code U8 ModemTrx4[] = {12, FREQ_CONTROL_INTE_12};

code U8 ModemTx1[] = {14, MODEM_DATA_RATE_2_14};
code U8 ModemTx2[] = {5, MODEM_TX_RAMP_DELAY_5};
```

Finally, the byte array (the properties) should be sent as-is by the bApi_SendCommand() function to initialize the radio as the following code section shows:

```
        // Set TRX parameters of the radio IC; data retrieved from the WDS-generated modem_params.h header
file
        bApi_SendCommand(ModemTrx1[0],&ModemTrx1[1]);      // Send API command to the radio IC
        bApi_WaitforCTS();                                 // Wait for CTS
        bApi_SendCommand(ModemTrx2[0],&ModemTrx2[1]);
        bApi_WaitforCTS();
        bApi_SendCommand(ModemTrx3[0],&ModemTrx3[1]);
        bApi_WaitforCTS();
        bApi_SendCommand(ModemTrx4[0],&ModemTrx4[1]);
        bApi_WaitforCTS();

        // Set TX parameters of the radio IC
        bApi_SendCommand(ModemTx1[0],&ModemTx1[1]);        // Send API command      to the radio IC
        bApi_WaitforCTS();                                 // Wait for CTS
        bApi_SendCommand(ModemTx2[0],&ModemTx2[1]);
        bApi_WaitforCTS();
```

Note: Some of the definitions in the modem_params.h are adequate for Transmit or Receive operation only (those are marked as “//all for TX in this section:” or “//all for RX below:” comments and should be used only for Transmit and Receive operation initialization). However, the properties defined in the section marked as “//all for general parameters in both TRX” have to be set for all the operation modes.

Several test cards are supported by the sample codes. The desired test card can be selected in *ezrp_next_api.h* (see "2.1. Test Card Options" on page 1). The desired operation mode can be set by the void *vSetPAMode(U8 bPaMode, U8 bModType);* function.

```
void vSetPAMode(U8 bPaMode, U8 bModType)
{
        abApi_Write[0] = CMD_SET_PROPERTY;
        abApi_Write[1] = PROP_PA_GROUP;
        abApi_Write[2] = 4;
        abApi_Write[3] = PROP_PA_MODE;
        abApi_Write[4] = PaSettings[bPaMode][0];
        abApi_Write[5] = PaSettings[bPaMode][1];
        abApi_Write[6] = PaSettings[bPaMode][2];
        if (bModType == MOD_OOK)
        {
        abApi_Write[7] = PaSettings[bPaMode][4];
        }
        else
        {
        abApi_Write[7] = PaSettings[bPaMode][3];
        }
        bApi_SendCommand(8, abApi_Write);
        vApi_WaitforCTS();
}
```

The function automatically gets its inputs by properly selecting the desired test card in *ezrp_next_api.h* and having the desired modulation in *modem_params.h*.

```
// Configuring PA mode
        vSetPAMode(EZRP_NEXT_TestCard,ModemTrx1[5]);          // Select proper testcard and modulation
```

3.3.3. Set Packet Content

The EZRadioPRO devices support four modes of operation:

- FIFO mode
- Packet handler mode
- Direct mode
- RAW data mode

Sample Code 1 uses Packet Handler mode. The preamble and sync word are configurable; both the length and the pattern are defined by the user. After configuring them, they are controlled automatically by the radio IC. The only thing the user application has to do with the packet (after the settings) is to put the payload into the transmit FIFO.

To configure packet parameters, such as preamble length and pattern, sync word length and pattern, fix packet length, and the bit order of the payload, properties must be written as shown below:

```
        // Set packet content
        // Set preamble length
        abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
        abApi_Write[1] = PROP_PREAMBLE_GROUP;               // Select property group
        abApi_Write[2] = 1;                                 // Number of properties to be
written
        abApi_Write[3] = PROP_PREAMBLE_TX_LENGTH;           // Specify property
        abApi_Write[4] = 0x05;                              // 5 bytes Tx preamble
        bApi_SendCommand(5,abApi_Write);                    // Send command to the radio IC
        vApi_WaitforCTS();                                  // Wait for CTS
        // Set preamble pattern
        abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
        abApi_Write[1] = PROP_PREAMBLE_GROUP;               // Select property group
        abApi_Write[2] = 1;                                 // Number of properties to be
written
        abApi_Write[3] = PROP_PREAMBLE_CONFIG;              // Specify property

        abApi_Write[4] = 0x31;                      // Use `1010` pattern, length defined in bytes
        bApi_SendCommand(5,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();                          // Wait for CTS
        // Set sync word
        abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
        abApi_Write[1] = PROP_SYNC_GROUP;           // Select property group
        abApi_Write[2] = 3;                         // Number of properties to be written
        abApi_Write[3] = PROP_SYNC_CONFIG;          // Specify property
        abApi_Write[4] = 0x01;                      // 2 bytes synch word: 0x2DD4
        abApi_Write[5] = 0xB4;                  // 1st sync byte: 0x2D; NOTE: LSB transmitted first!
        abApi_Write[6] = 0x2B;                  // 2nd sync byte: 0xD4; NOTE: LSB transmitted first!
        bApi_SendCommand(7,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();                          // Wait for CTS
        // General packet config (set bit order)
        abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
        abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
        abApi_Write[2] = 1;                         // Number of properties to be written
        abApi_Write[3] = PROP_PKT_CONFIG1;          // Specify property
        abApi_Write[4] = 0x00;                      // payload data goes MSB first
        bApi_SendCommand(5,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();                          // Wait for CTS
```

The actual packet content has to be filled into the Transmit FIFO each time before packet transmission:

```
// Put the payload to Tx FIFO
abApi_Write[0] = 0x42;                  // write 0x42 ('B')
abApi_Write[1] = 0x55;                  // write 0x55 ('U')
abApi_Write[2] = 0x54;                  // write 0x54 ('T')
…                                       …
abApi_Write[7] = 0x0D;                  //write 0x0D (CR)

bApi_WriteTxDataBuffer(0x08, &abApi_Write[0]);      // Write data to Tx FIFO
bApi_WaitforCTS();                                  // Wait for CTS
```

3.3.4. 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. Setting the modulation type can be performed via a property, *MODEM_MOD_TYPE*. In the sample codes, this setting is included in the definition of the *modem_params.h* header file generated by WDS.

Note: The EZRadioPRO devices can also be set to provide non-modulated carrier signals or PN9 random data modulated signals for test purposes.

3.3.4.1. 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 $f_0 + \Delta f_{FSK}$.

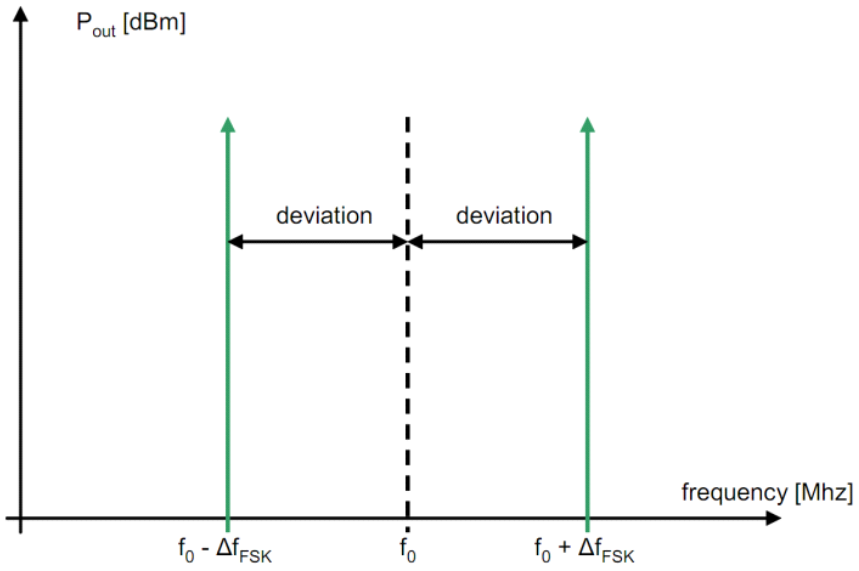


Figure 18. FSK Modulation

FSK modulation is resistant to interference; 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 the signal frequency falls within the receiver's filter bandwidth. Increasing the signal bandwidth results in a decrease in system sensitivity.

3.3.4.2. 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.

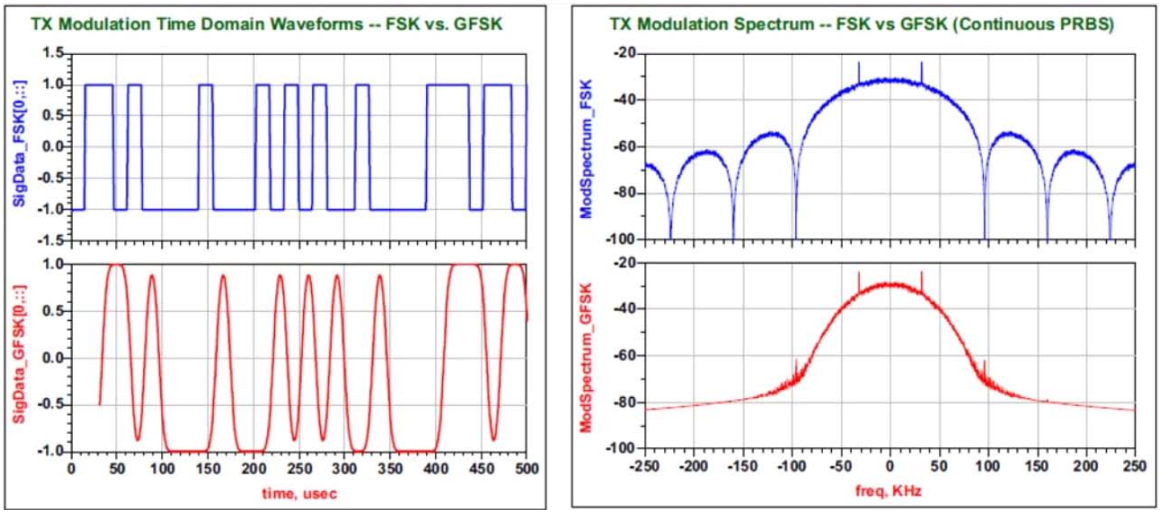


Figure 19. FSK and GFSK Modulation Differences (Time Domain and Spectrum)

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.

3.3.4.3. On-Off Keying

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

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

3.3.5. Crystal Oscillator Tuning Capacitor

The accuracy of the radio center frequency is affected 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 through the API by tuning the capacitance bank of the crystal internally.

The crystal capacitance bank can be used to compensate for crystal mismatch by tuning it to the proper capacitance value. Assuming that the same PCB and type of crystal is used during the entire life cycle of the product, the crystal 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, and is set in the sample codes per the examples below.

```
#define CAP_BANK_VALUE          0x48        // Capacitor bank value for adjusting the XTAL frequency
…                                           …
// Adjust XTAL clock frequency
abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
abApi_Write[1] = PROP_GLOBAL_GROUP;         // Select property group
abApi_Write[2] = 1;                         // Number of properties to be written
abApi_Write[3] = PROP_GLOBAL_XO_TUNE;       // Specify property
abApi_Write[4] = CAP_BANK_VALUE;            // Set cap bank value to adjust XTAL clock frequency
bApi_SendCommand(5,abApi_Write);            // Send command to the radio IC
vApi_WaitforCTS();                          // Wait for CTS
```

3.3.6. Send a Packet

After the MCU and the radio are initialized for packet transmission, the firmware sets the radio into SLEEP mode by issuing a *CHANGE_STATE* command. It is very important to avoid any SPI activity after issuing the command; otherwise, the radio wakes up from SLEEP mode automatically.

```
//Put the Radio into SLEEP state
abApi_Write[0] = CMD_CHANGE_STATE;          // Change state command
abApi_Write[1] = 0x01;                      // SLEEP state
bApi_SendCommand(2,abApi_Write);            // Send command to the radio IC
```

The firmware continuously monitors the status of the push buttons in the main loop. Once a button is pressed, the software wakes up the radio from SLEEP mode, fills the appropriate payload into the transmit FIFO, starts the packet transmission, and waits for the packet sent interrupt. During packet transmission, one of the LEDs is turned on. In order to actually transmit, the *START_TX* command must be sent to the radio IC. Only the packet sent interrupt is enabled; so, it is enough to monitor the NIRQ pin to determine when the packet is sent. After packet transmission, the interrupt status registers are read to release the NIRQ pin, and the radio is set to SLEEP mode.

Note: In simple Packet Handler mode, the TX_LEN input parameter of the START_TX command defines the number of transmitted bytes after the synchron word.

The following code section shows how a button press is captured and how the packet is transmitted:

```
if (bButtonPushTrack)                                   // Check if any of the buttons was pushed
{
        while((GetPB(1)==0) || (GetPB(2)==0) || (GetPB(3)==0) || (GetPB(4)==0)); // Wait until released

        //wake up the radio from SLEEP mode
        abApi_Write[0] = CMD_CHANGE_STATE;              // Change state command
        abApi_Write[1] = 0x02;                          // SPI active state
        bApi_SendCommand(2,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();

        // Put the payload to Tx FIFO
        abApi_Write[0] = 0x42;                          // write 0x42 ('B')
        abApi_Write[1] = 0x55;                          // write 0x55 ('U')
        abApi_Write[2] = 0x54;                          // write 0x54 ('T')
        abApi_Write[3] = 0x54;                          // write 0x54 ('T')
        abApi_Write[4] = 0x4F;                          // write 0x4F ('O')
        abApi_Write[5] = 0x4E;                          // write 0x4E ('N')
        switch (bButtonPushTrack)           // write button nmbr and switch off the corresponding LED
        {
                case 1:
                        abApi_Write[6] = 0x31;
                        SetLed(1);
                break;
                case 2:
                        abApi_Write[6] = 0x32;
                        SetLed(2);
                break;
                case 4:
                        abApi_Write[6] = 0x33;
                        SetLed(3);
                break;
                case 8:
                        abApi_Write[6] = 0x34;
                        SetLed(4);
                break;
        }
        abApi_Write[7] = 0x0D;                                  //write 0x0D (CR)
        bApi_WriteTxDataBuffer(0x08, &abApi_Write[0]);          // Write data to Tx FIFO
        vApi_WaitforCTS();                                      // Wait for CTS

        // Start Tx
        abApi_Write[0] = CMD_START_TX;  // Use Tx Start command
        abApi_Write[1] = 0;                             // Set channel number
        abApi_Write[2] = 0x10;                          // Sleep state after Tx, start Tx immediately
        abApi_Write[3] = 0x00;                          // 8 bytes to be transmitted (fix packet length, no packet field used)
        abApi_Write[4] = 0x08;                          // 8 bytes to be transmitted (fix packet length, no packet field used)
        bApi_SendCommand(5,abApi_Write);  // Send command to the radio IC

        // Wait for packet sent interrupt
        while(EZRP_NIRQ == 1);

        // Read ITs, clear pending ones
        abApi_Write[0] = CMD_GET_INT_STATUS;    // Use interrupt status command
        abApi_Write[1] = 0;                             // Clear PH_CLR_PEND
        abApi_Write[2] = 0;                             // Clear MODEM_CLR_PEND
        abApi_Write[3] = 0;                             // Clear CHIP_CLR_PEND
```

```
        bApi_SendCommand(4,abApi_Write);          // Send command to the radio IC
        bApi_GetResponse(8,abApi_Read);           // Make sure that CTS is ready then get the response

        //Put the Radio into SLEEP state
        abApi_Write[0] = CMD_CHANGE_STATE;        // Change state command
        abApi_Write[1] = 0x01;                    // SLEEP state
        bApi_SendCommand(2,abApi_Write);          // Send command to the radio IC

        for (wDelay=0; wDelay<30000; wDelay++);   // Wait to show LED combination

        switch (bButtonPushTrack)                 // check wich button was pressed and then clear the proper
LED
        {
                case 1:
                        ClearLed(1);
                break;
                case 2:
                        ClearLed(2);
                break;
                case 4:
                        ClearLed(3);
                break;
                case 8:
                        ClearLed(4);
                break;
        }
        bButtonPushTrack = 0;                     // Clear button push track variable
}
```

3.4. Packet Reception Using the Packet Handler(Sample Code 1)

The software example works as a continuous receiver node. After the MCU and the EZRadioPRO receiver device are 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 Figure 14 on page 16, it blinks the appropriate LED. If a packet is received with a payload “Button1”, the software blinks LED1 on the development board (LED2 for “Button2”, etc.). After a successful packet reception, the software restarts the receiver and waits for the next packet. Figure 20 shows the flowchart of the receive side of Sample Code 1.

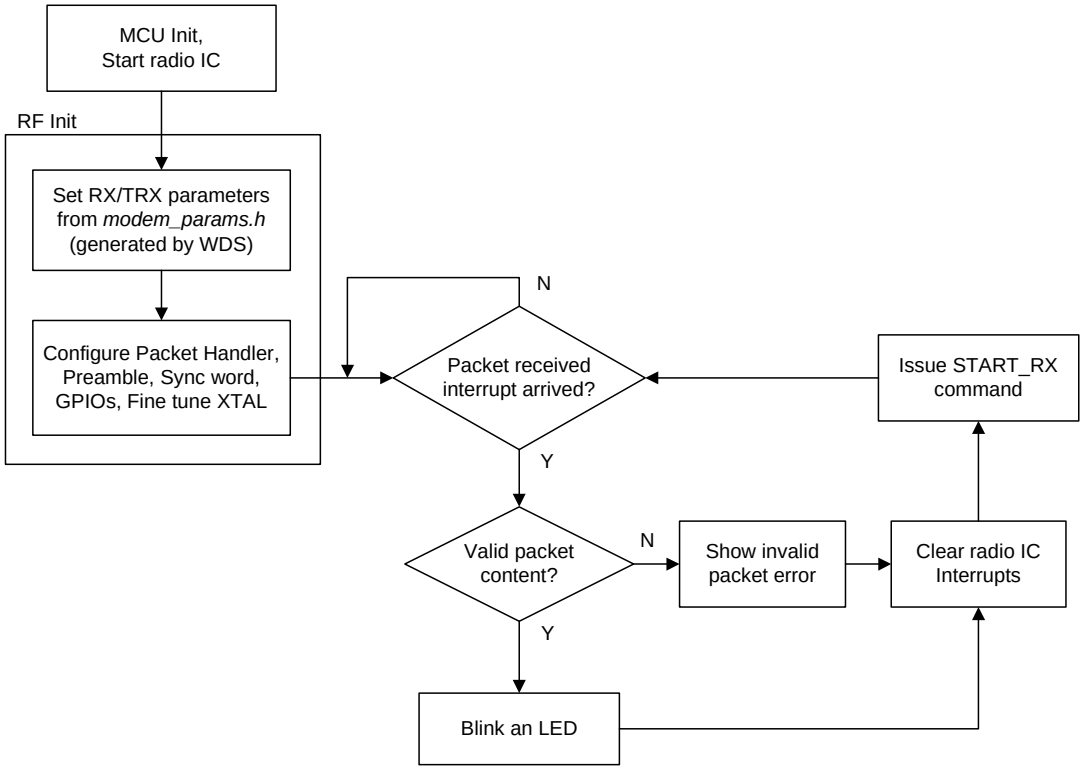


Figure 20. Receiver Side Flowchart

As described earlier, Sample Code 1 uses the chip in Packet Handler mode. From a receiver point of view, this means that the radio recognizes the preamble and the sync word automatically. After detecting the synchron word, the radio stores the predefined number of bytes in the Receive FIFO. The MCU can read the payload by the RX_FIFO_READ command.

Initialization of the radio IC when set as a receiver node is the same as when it is set as a transmitter node. First, the MCU needs to be initialized. This is followed by resetting the radio IC and waiting for the device to turn on. Then, the very first command, POWER_UP, should be sent to the radio IC to invoke the API. For more details, about the initialization see "3.3.1. Start the Radio" on page 16.

3.4.1. Preamble Detection

It is mandatory to start every packet with a preamble; the only exception is when the device is used in raw data mode. 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. Ideally, the preamble is alternating ones and zeroes (“0101”). To enable the easy replacement of existing solutions, the Si446x family supports any non-standard preamble pattern up to four bytes and, optionally, may allow bit errors.

The EZRadioPRO receivers have a built-in Preamble Detector. After settling the receiver, the radio compares the received bits to the previously-set preamble bit pattern. If the preamble detector finds the predefined length of consecutive preamble bits in the received data, the radio IC reports a valid preamble and generates a “preamble detect” interrupt for the host MCU.

The preamble detection threshold 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) contains a short preamble bit pattern. In this case, a longer preamble threshold (e.g., 20 bits) is recommended. If the receiver is enabled synchronously just before the packet is transmitted, a short preamble detection threshold (e.g. eight 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 at least 20 bits when using antenna diversity. Likewise, when using the AFC in the receiver, additional measurements are required during the preamble reception resulting in a longer required preamble length. For recommended transmit preamble length and preamble detection threshold, refer to the data sheet.

```
//Set packet content
//Set preamble length
abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
abApi_Write[1] = PROP_PREAMBLE_GROUP;               // Select property group
abApi_Write[2] = 1;                                 // Number of properties to be written
abApi_Write[3] = PROP_PREAMBLE_CONFIG_STD_1;        // Specify property
abApi_Write[4] = 20;                                // 20 bits preamble detection threshold
bApi_SendCommand(5,abApi_Write);                    // Send API command to the radio IC
vApi_WaitforCTS();                                  // Wait for CTS

// Set preamble pattern
abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
abApi_Write[1] = PROP_PREAMBLE_GROUP;               // Select property group
abApi_Write[2] = 1;                                 // Number of properties to be written
abApi_Write[3] = PROP_PREAMBLE_CONFIG;              // Specify property
abApi_Write[4] = 0x31;                              // Use `1010` pattern, length defined in bytes
bApi_SendCommand(5,abApi_Write);                    // Send API command to the radio IC
vApi_WaitforCTS();                                  // Wait for CTS
```

The radio can raise the *PREAMBLE_DETECT* interrupt flag if the preamble is successfully detected. If it is enabled, the radio pulls low the NIRQ pin, which can be released by reading the *MODEM_STATUS* pending register.

3.4.2. Sync Word Detection

After the preamble is detected, the radio looks for the sync word. This known pattern can be up to four bytes and fully configurable. If the radio does not find the sync word within timeout, it goes into a state defined by the *RXTIMEOUT_STATE* input parameter of the *START_RX* command. The timeout is defined in bit timing and is as long as the length of the sync word plus four bit times.

```
// Set sync word
abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
abApi_Write[1] = PROP_SYNC_GROUP;                   // Select property group
abApi_Write[2] = 3;                                 // Number of properties to be written
abApi_Write[3] = PROP_SYNC_CONFIG;                  // Specify property
abApi_Write[4] = 0x01;                              // 2 bytes synch word: 0x2DD4
abApi_Write[5] = 0xB4;                              // 1st sync byte: 0x2D; NOTE: LSB transmitted first!
abApi_Write[6] = 0x2B;                              // 2nd sync byte: 0xD4; NOTE: LSB transmitted first!
bApi_SendCommand(7,abApi_Write);                    // Send command to the radio IC
vApi_WaitforCTS();                                  // Wait for CTS
```

Note: The sync word is always transmitted LSB first.

If the sync word is detected, the radio can generate *SYNCH_DETECT* interrupt and pull low NIRQ if it is enabled.

After the sync word is detected the radio receives the predefined amount of bytes and fills them into the Receive FIFO. In simple Packet Handler mode, when no FIELD is used, the *RX_LEN* parameter of the *START_RX* command defines how many bytes the radio need to receive after the sync word.

3.4.3. Receiving the Payload

After configuring the MCU and the radio, the example code goes into a continuous loop and waits for packet reception. It then reads the contents of the Receive FIFO and compares the received payload to the expected data. If a button press is initiated, one of the LEDs blinks on the transmit side, and the same LED should blink on the receiver side. If a corrupted packet is received, all four LEDs turns on.

The MCU clears all pending interrupts and sets the radio into receive mode.

The radio is capable of measuring the signal strength (RSSI) of the incoming signal. It is also capable of saving the RSSI upon predefined events, like preamble, sync word detection, etc. The fastest way to read the latched RSSI is to use the FRRs (Fast Response Registers). The code example configures the radio to latch the actual RSSI after detecting the sync word, and the MCU reads the value right after the packet is received using the FRRs.

```
// Wait for detect interrupt
if(EZRP_NIRQ == 0)
{
        // Read PH IT registers to see the cause for the IT
        abApi_Write[0] = CMD_GET_PH_STATUS;             // Use packet handler status command
        abApi_Write[1] = 0x00;                          // dummy byte
        bApi_SendCommand(2,abApi_Write);        // Send command to the radio IC
        bApi_GetResponse(1,abApi_Read);         // Get the response
        if((abApi_Read[0] & 0x10) == 0x10)      // Check if packet received
        {//Packet received
                // Get RSSI
                bApi_GetFastResponseRegister(CMD_FAST_RESPONSE_REG_C,1,abApi_Read);
                // Read the FIFO
                bApi_ReadRxDataBuffer(8,abApi_Read);
                fValidPacket = 0;
                // Check the packet content
                if ((abApi_Read[0]=='B') && (abApi_Read[1]=='U') && (abApi_Read[2]=='T') &&
(abApi_Read[3]=='T') && (abApi_Read[4]=='O') && (abApi_Read[5]=='N'))
                {
                        bButtonNumber = abApi_Read[6] & 0x07;  // Get button info
                        if((bButtonNumber > 0) && (bButtonNumber < 5))
                        {
                                SetLed(bButtonNumber);                  // Turn on the appropriate LED
                                for(wDelay=0; wDelay<30000; wDelay++);          // Wait to show LED
                                ClearLed(bButtonNumber);                // Turn off the corresponding LED
                                fValidPacket = 1;
                        }
                }
                // Packet content is not what was expected
                if(fValidPacket == 0)
```

```
        {
                        SetLed(1);                                      // Blink all LEDs
                        SetLed(2);
                        SetLed(3);
                        SetLed(4);
                        for(wDelay=0; wDelay<30000; wDelay++);
                        ClearLed(1);
                        ClearLed(2);
                        ClearLed(3);
                        ClearLed(4);
                }
        }

        // Read ITs, clear pending ones
        abApi_Write[0] = CMD_GET_INT_STATUS;        // Use interrupt status command
        abApi_Write[1] = 0;                         // Clear PH_CLR_PEND
        abApi_Write[2] = 0;                         // Clear MODEM_CLR_PEND
        abApi_Write[3] = 0;                         // Clear CHIP_CLR_PEND
        bApi_SendCommand(4,abApi_Write);            // Send command to the radio IC
        bApi_GetResponse(8,abApi_Read);             // Make sure that CTS is ready then get the response

        // Start Rx
        abApi_Write[0] = CMD_START_RX;                  // Use start Rx command
        abApi_Write[1] = 0;                             // Set channel number
        abApi_Write[2] = 0x00;                          // Start Rx immediately
        abApi_Write[3] = 0x00;                          // 8 bytes to receive
        abApi_Write[4] = 0x08;                          // 8 bytes to receive
        abApi_Write[5] = 0x00;                          // No change if Rx timeout
        abApi_Write[6] = 0x03;                          // Ready state after Rx
        abApi_Write[7] = 0x03;                          // Ready if Rx invalid
        bApi_SendCommand(8,abApi_Write);                // Send API command to the radio IC
        vApi_WaitforCTS();                              // Wait for CTS
}
```

3.5. Using the FIELDs (Sample Code 2)

This section focuses on how to use the packet fields when using the packet handler in the Si446x. Packet fields can be used to assign individual properties either to certain parts or to the entire payload. Maximum five fields can be defined; it is up to the user how to split up the payload for these fields. The payload may consist of one field or more; each of them may have different configurations, such as CRC calculation, data whitening, or Manchester coding. The transmit/receive FIFOs are a maximum of 64 bytes long; the length of a field is a maximum of 255 bytes. It is possible to use a field that is larger than the FIFO. In such a case, the FIFO writing/reading should be performed in multiple cycles using the FIFO almost full/empty interrupts.

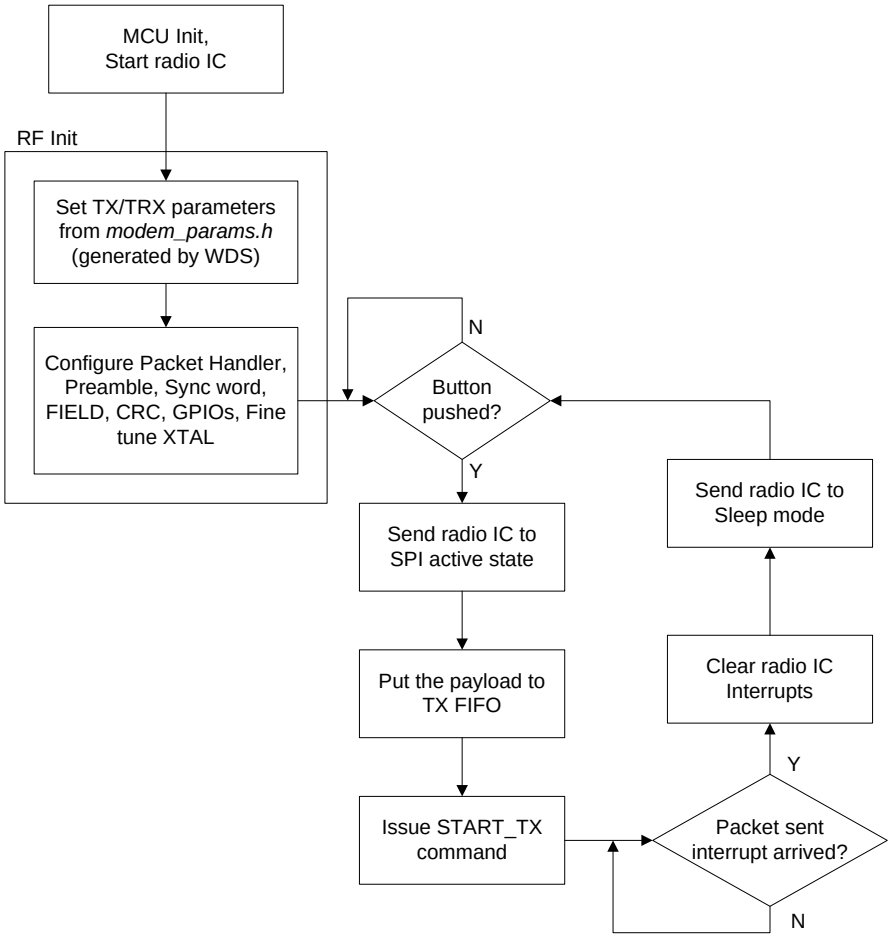


Figure 21. Flowchart of Sample Code 2 (Transmitter Side)

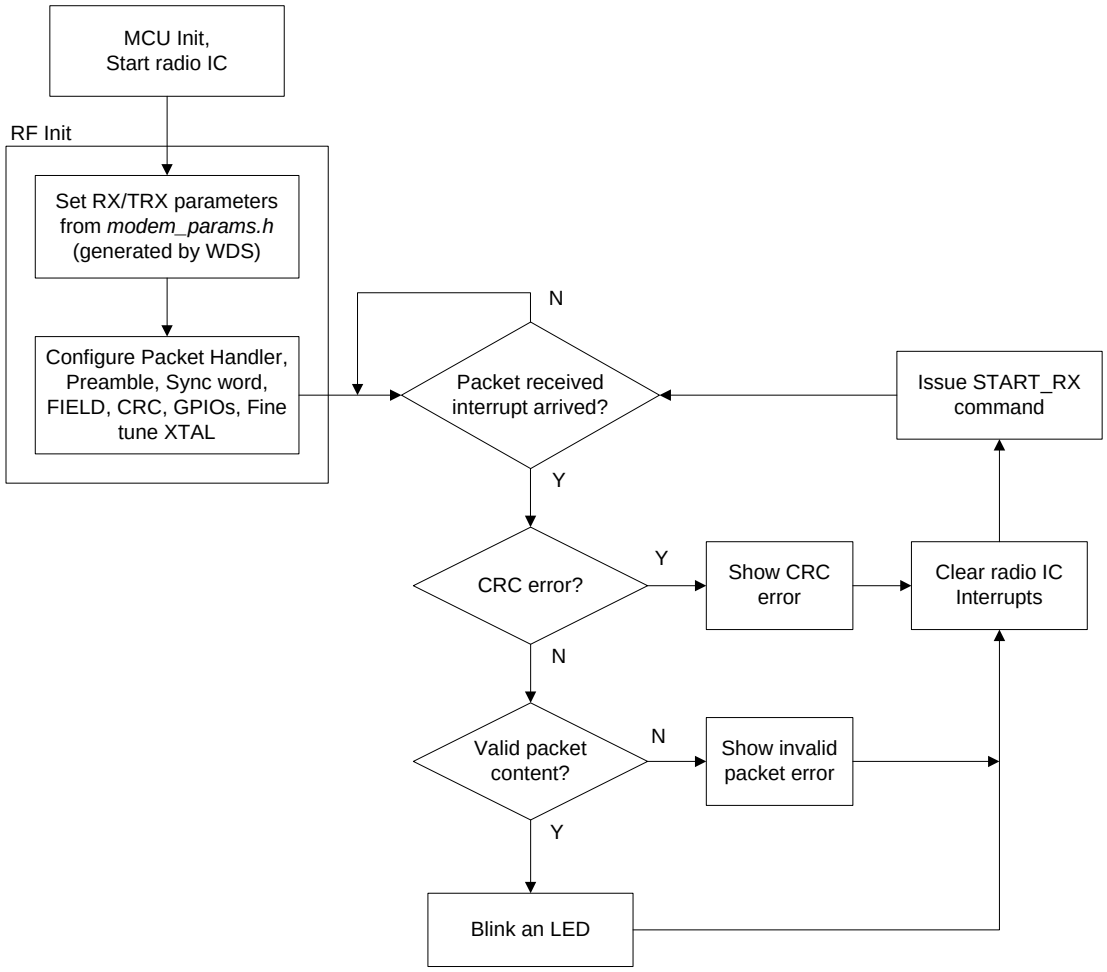


Figure 22. Flowchart of Sample Code 2 (Receiver Side)

Sample Code 2 has the same functionality as Sample Code 1 except that it uses CRC check and therefore a FIELD. The FIELD covers the whole 8-byte payload, and two CRC bytes are attached to the end of the field in the packet. Note that even though preamble, sync word, and CRC are also included in the packet, the user only has to write/read the receive/transmit FIFO to send/get the actual payload. The packet handler takes care of everything else, such as synchronizing the clock frequencies with the preamble, recognizing the packet start with the sync word, and calculating and checking for CRC error. The CRC calculation starts at the beginning of the field, and it is transmitted (or checked) at the end of it. In the sample code, the following packet is used:

Preamble					Sync word		Payload: `BUTTON1` + CR								CRC	
AA	AA	AA	AA	AA	2D	D4	42	55	54	54	4F	4E	31	0D	7A	B1
5 bytes					2 bytes		8 bytes									

Configuring the device to use one field is implemented as the following:

Tx side:

```
// Set packet fields (use only one field out of the available five)
abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
abApi_Write[1] = PROP_PKT_GROUP;                    // Select property group
abApi_Write[3] = PROP_PKT_FIELD_1_LENGTH_12_8;      // Specify first property
abApi_Write[2] = 4;                                 // Number of properties to be written
abApi_Write[4] = 0x00;
abApi_Write[5] = 0x08;                              // 8 byte long packet field
abApi_Write[6] = 0x00;                              // No 4GFSK/whitening/Manchester coding for this
field
abApi_Write[7] = 0xA2;                              // Start CRC calc. from this field, check CRC at the
end
bApi_SendCommand(8,abApi_Write);                    // Send command to the radio IC
vApi_WaitforCTS();                                  // Wait for CTS
```

Rx side:

```
// Set packet fields (use only one field out of the available five)
abApi_Write[0] = CMD_SET_PROPERTY;                  // Use property command
abApi_Write[1] = PROP_PKT_GROUP;                    // Select property group
abApi_Write[2] = 4;                                 // Number of properties to be written
abApi_Write[3] = PROP_PKT_FIELD_1_LENGTH_12_8;      // Specify first property
abApi_Write[4] = 0x00;
abApi_Write[5] = 0x08;                              // 8 byte long packet field
abApi_Write[6] = 0x00;                              // No 4GFSK/whitening/Manchester coding for this
field
abApi_Write[7] = 0x8A;                              // Start CRC calc. from this field, check CRC at the
end
bApi_SendCommand(8,abApi_Write);                    // Send command to the radio IC
vApi_WaitforCTS();                                  // Wait for CTS
```

When using field(s) the packet length should be set to zero in the *START_TX* and *START_RX* commands (length should be defined in *PKT_FIELD_x_LENGTH_x_x* properties instead):

```
// Start Tx
abApi_Write[0] = CMD_START_TX;                      // Use Tx Start command
abApi_Write[1] = 0;                                 // Set channel number
abApi_Write[2] = 0x30;                              // READY state after Tx, start Tx immediately
abApi_Write[3] = 0x00;                              // packet fields used, do not enter packet length here
abApi_Write[4] = 0x00;                              // packet fields used, do not enter packet length here
bApi_SendCommand(5,abApi_Write);                    // Send command to the radio IC
vApi_WaitforCTS();                                  // Wait for CTS
```

```
// Start Rx
abApi_Write[0] = CMD_START_RX;                  // Use start Rx command
abApi_Write[1] = 0;                             // Set channel number
abApi_Write[2] = 0x00;                          // Start Rx immediately
abApi_Write[3] = 0x00;                          // Packet fields used, do not enter packet length here
abApi_Write[4] = 0x00;                          // Packet fields used, do not enter packet length here
abApi_Write[5] = 0x00;                          // Rx state if Rx timeout
abApi_Write[6] = 0x03;                          // READY state after Rx
abApi_Write[7] = 0x03;                          // READY state if Rx invalid
bApi_SendCommand(8,abApi_Write);                // Send API command to the radio IC
vApi_WaitforCTS();
```

With the packet fields feature, the packet structure is quite flexible. For instance, if four fields are used, it is possible that the CRC calculation starts at the beginning of the first field; for the second field, it is completely disabled, and it is continued in the third field. At the end of the third field, the CRC check sum is transmitted/checked; for the fourth field, even a separate CRC check may be applied. Like CRC calculation, data whitening and Manchester coding can also be applied individually for each field.

It is also possible to use variable packet length with the Si446x. In this case, one of the fields has variable length, while all the other fields should have a fixed length. It is important that, on the receiver side, one fixed-length field should be used before the variable length field, where the last byte of the fixed-length field defines the actual length of the variable field. As for the transmitter, the packet should contain the length of the packet and should be entered “manually” as part of the payload. On the receiver side, the length is extracted from the received packet automatically. See "3.7. Variable Packet Length (Sample Code 3)" on page 42 for detailed information on using variable packet length.

The fields can be set up through properties. Each field has four configuration properties:

- PKT_FIELD_x_LENGTH_12_8
- PKT_FIELD_x_LENGTH_7_0
- PKT_FIELD_x_CONGIF
- PKT_FIELD_x_CRC_CONFIG

If the fields are used, the actual lengths of the transmit/receive packets are defined by the field length properties (i.e. PKT_FIELD_x_LENGTH_12_8, PKT_FIELD_x_LENGTH_7_0), and NOT by the TX_LEN / RX_LEN input parameters of the START_TX / START_RX commands (TX_LEN / RX_LEN must be zeroes in such cases). Sharing these properties between transmit and receive mode operations may require changing them frequently if the radio IC is used for bidirectional communication. In order to ensure fast switching between transmit and receive modes, there is an option to avoid this sharing by doubling the field configuration properties in the API:

- Properties starting with PKT_FIELD_ can be used only for transmit operation.
- Properties starting with PKT_RX_FIELD_ can be used only for receiver operation.

To select this option, there is a control bit in the *PKT_CONFIG1* property:

- If the PH_FIELD_SPLIT bit is set, then the PKT_FIELD_ properties are used for Transmit operation, while PKT_RX_FIELD_ properties are used for Receive operation. Using this method, fast turnaround time can be achieved since both Transmit and Receive operations could fully be configured prior to entering Receive mode or starting packet transmission.
- If the PH_FIELD_SPLIT bit is cleared, then the PKT_FIELD_ properties define both the Transmit and Receive operations. This can be useful for single way operation.

Sample Code 2 focuses on a simple field usage where one field is used with fix length (8 bytes) and CRC16 calculation. It uses only one field on both the Transmit and Receive sides; the field is configured through only PKT_FIELD_ properties.

```
// General packet config (set bit order)
        abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
        abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
        abApi_Write[2] = 1;                         // Number of properties to be written
        abApi_Write[3] = PROP_PKT_CONFIG1;          // Specify property
        abApi_Write[4] = 0x00;                      // payload data goes MSB first
        bApi_SendCommand(5,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();                          // Wait for CTS

        // Set packet fields (use only one field out of the available five)
        abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
        abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
        abApi_Write[2] = 4;                         // Number of properties to be written
        abApi_Write[3] = PROP_PKT_FIELD_1_LENGTH_12_8;    // Specify first property
        abApi_Write[4] = 0x00;
        abApi_Write[5] = 0x08;                      // 8 byte long packet field
        abApi_Write[6] = 0x00;                      // No 4GFSK/whitening/Manchester coding for this
field
        abApi_Write[7] = 0x8A;                      // Start CRC calc. from this field, check CRC at the
end
        bApi_SendCommand(8,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();                          // Wait for CTS

        // Configure CRC polynomial and seed
        abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
        abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
        abApi_Write[2] = 1;                         // Number of properties to be written
        abApi_Write[3] = PROP_PKT_CRC_CONFIG;       // Specify property
        abApi_Write[4] = 0x05;                      // CRC seed: all `1`s, poly: No. 5, 16bit
        bApi_SendCommand(5,abApi_Write);            // Send command to the radio IC
        vApi_WaitforCTS();                          // Wait for CTS
```

The transmitter code has the same functionality, and it is almost the same as the simple FIFO transmit mode (except the field settings); therefore, it is not detailed in this chapter.

The receiver code works the same way as the simple FIFO Receiver example, except that it checks the CRC during packet reception. After a packet is received, not only the packet valid, but the CRC error interrupt is checked also. If CRC error occurs, then Receive FIFO is not read, but the pending interrupts are cleared, and the radio is set to receive mode again. Also, the MCU blinks LED1 and LED4 for a short period of time to show the user the CRC error. The radio raises the CRC2_ERROR interrupt and pulls the NIRQ pin low in case of CRC error.

If a packet is received with correct CRC, then the PACKET_RX interrupt is set and the MCU reads the payload from the Receive FIFO. It then compares the received data with the expected packet content and blinks the LED accordingly.

```
// Wait until nIRQ goes low
if(EZRP_NIRQ == 0)
{
        // Read PH IT registers to see the cause for the IT
        abApi_Write[0] = CMD_GET_PH_STATUS;          // Use packet handler status command
        abApi_Write[1] = 0x00;                       // dummy byte
        bApi_SendCommand(2,abApi_Write);             // Send command to the radio IC
        bApi_GetResponse(1,abApi_Read);          // Make sure that CTS is ready then get the response

        // Check CRC error
        if((abApi_Read[0] & 0x0C) != 0x00 )
        {
                SetLed(1);                           // Show LED combination for CRC error
                SetLed(4);
        }
        // Check Packet valid interrupt
        if((abApi_Read[0] & 0x10) == 0x10)           // Check if packet received
        {//Packet received
                // Get RSSI
                bApi_GetFastResponseRegister(FAST_RESPONSE_REG_C,1,abApi_Read);
                // Read the FIFO
                bApi_ReadRxDataBuffer(8,abApi_Read);

                fValidPacket = 0;
                // Check the packet content
                if ((abApi_Read[0]=='B') && (abApi_Read[1]=='U') && (abApi_Read[2]=='T') &&
(abApi_Read[3]=='T') && (abApi_Read[4]=='O') && (abApi_Read[5]=='N'))
```

```
                {
                        bButtonNumber = abApi_Read[6] & 0x07;           // Get button info
                        if((bButtonNumber > 0) && (bButtonNumber < 5))
                        {
                                SetLed(bButtonNumber);  // Turn on the appropriate LED
                                for(wDelay=0; wDelay<30000; wDelay++);          // Wait to show LED
                                ClearLed(bButtonNumber);// Turn off the corresponding LED
                                fValidPacket = 1;
                        }
                }
                // Packet content is not what was expected
                if(fValidPacket == 0)
                {
                        SetLed(1);                      // Blink all LEDs
                        SetLed(2);
                        SetLed(3);
                        SetLed(4);
                }
        }
        for(wDelay=0; wDelay<30000; wDelay++);
        ClearLed(1);
        ClearLed(2);
        ClearLed(3);
        ClearLed(4);

        // Read ITs, clear pending ones
        abApi_Write[0] = CMD_GET_INT_STATUS;    // Use interrupt status command
       abApi_Write[1] = 0;                      // Clear PH_CLR_PEND
        abApi_Write[2] = 0;                     // Clear MODEM_CLR_PEND
        abApi_Write[3] = 0;                     // Clear CHIP_CLR_PEND
        bApi_SendCommand(4,abApi_Write);        // Send command to the radio IC
        bApi_GetResponse(8,abApi_Read);         // Make sure that CTS is ready then get the response

        // Start Rx
        abApi_Write[0] = CMD_START_RX;          // Use start Rx command
        abApi_Write[1] = 0;                     // Set channel number
        abApi_Write[2] = 0x00;                  // Start Rx immediately
        abApi_Write[3] = 0x00;                  // packet fields used, do not enter packet length here
        abApi_Write[4] = 0x00;                  // packet fields used, do not enter packet length here
        abApi_Write[5] = 0x00;                  // Rx state if Rx timeout
        abApi_Write[6] = 0x03;                  // READY state after Rx
        abApi_Write[7] = 0x03;                  // READY state if Rx invalid
        bApi_SendCommand(8,abApi_Write);        // Send API command to the radio IC
vApi_WaitforCTS();
}
```

3.6. Bidirectional Communication (Sample Code 3)

Sample Code 3 demonstrates how to use the Si446x for a two-way link communication. By default, the – two – devices are in receiver mode and waiting for either a packet reception, or a button push. When pressing PB1 on one of them, it sends a packet to the other device, which blinks its LED1 to show packet reception. Then the receiver device sends back an acknowledgement packet to the sender, which is indicated by LED2.

The flowchart of Sample Code 3 is shown in Figure 23.

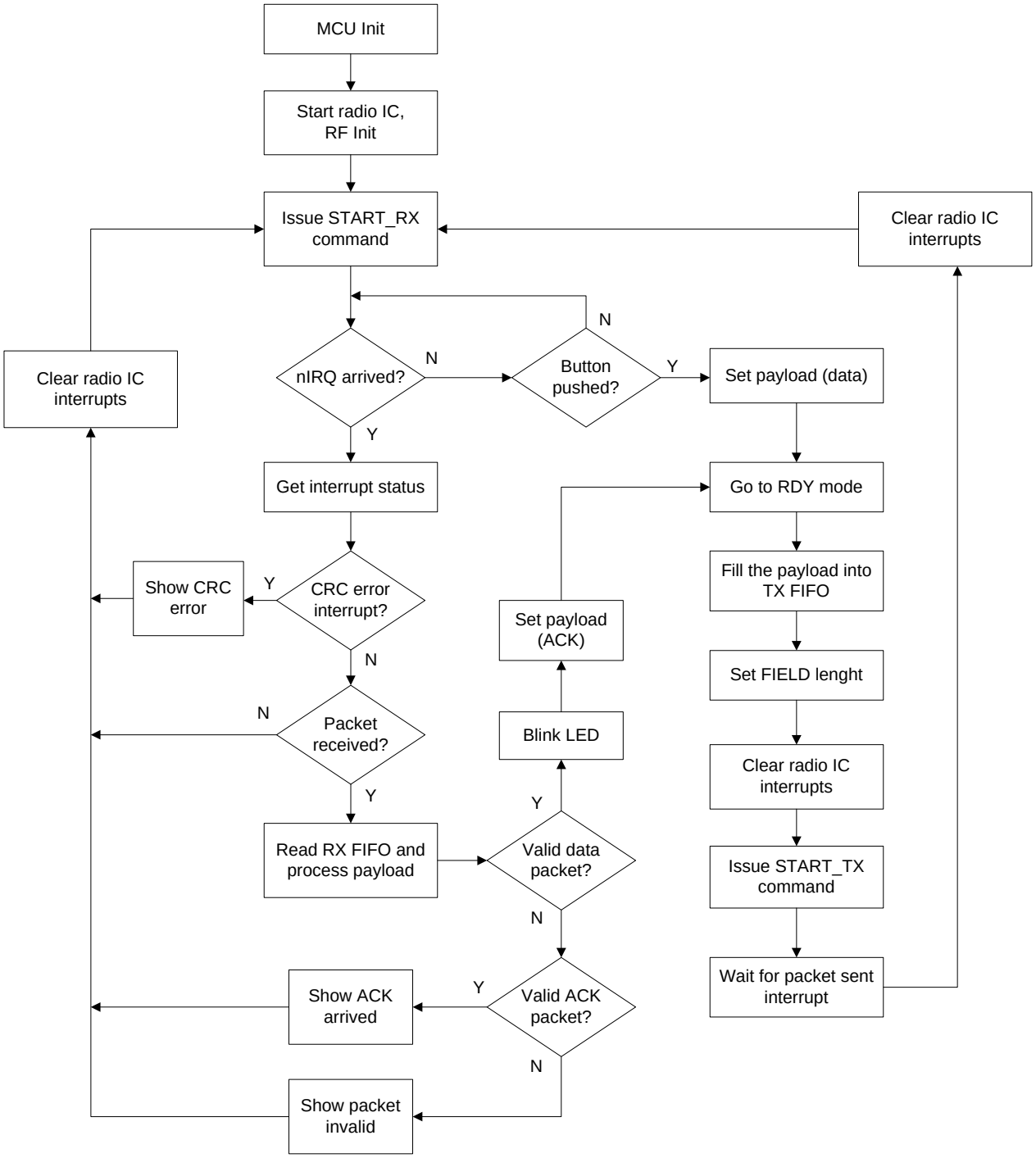


Figure 23. Sample Code 3 Flowchart

This sample code shows how to use the variable field length. There is no automatic operation on the Transmit side in terms of using variable packet length; the transmitted packet has to be set up by the host MCU, and the length field has to be placed at the proper place of the packet. Therefore, using only one field on the transmit side is sufficient:

- The packet content shall be composed by the host MCU.
- The packet needs to be filled into the FIFO.
- The length of the transmitted packet shall be set by the PKT_FIELD_1_LENGTH_12_8 and PKT_FIELD_1_LENGTH_7_0 properties.

In order to send single data or acknowledgement packets, the Si446x should change its state. In the code example, the device first goes to Ready mode by calling CHANGE_STATE command, so that it will not to receive any further packets while transmitting or preparing for transmitting:

```
// Go to Ready mode
abApi_Write[0] = CMD_CHANGE_STATE;        // Use change state command
abApi_Write[1] = 3;                       // Go to Ready mode
bApi_SendCommand(2,abApi_Write);          // Send command to the radio IC
vApi_WaitforCTS();                        // Wait for CTS
```

Receiving the data is implemented in a similar manner as in Sample Code 1 and 2. First, the latched RSSI information is read out, then the length of the payload by using the FIFO_INFO API command, and, finally, based on the extracted length information, the data bytes are read out from the FIFO:

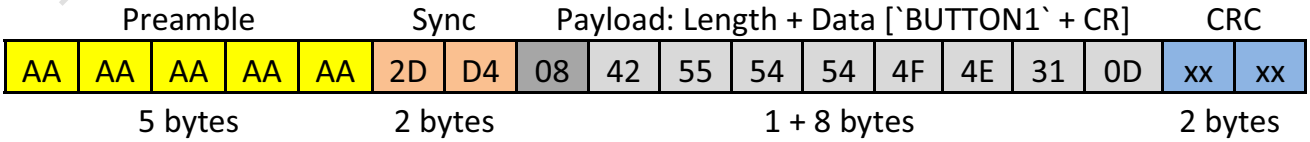
```
// Get RSSI
bApi_GetFastResponseRegister(FAST_RESPONSE_REG_C,1,&bRssi);
// Get length of the packet
abApi_Write[0] = CMD_FIFO_INFO;           // Use FIFO info command to get the length info
bApi_SendCommand(1,abApi_Write);          // Send command to the radio IC
bApi_GetResponse(2,&abApi_Read[0]);       // Make sure that CTS is ready then get the response
// Read out the payload after the length
bApi_ReadRxDataBuffer(abApi_Read[0],&abApi_Read[2]);
```

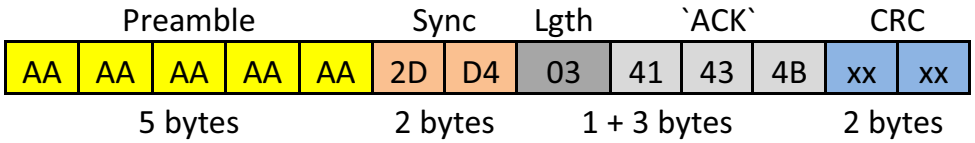
For transmitting the packet, a separate function, bSendPacket(), has been written. Its input parameters are the length of the data to be transmitted and a pointer showing the memory location where the data starts. The function is responsible for doing the state changes from Receive → Ready → Transmit, writing the transmit FIFO, transmitting with the *START_TX* command, and setting the actual packet field lengths. After packet transmission the radio goes into Ready mode.

```
U8 bSendPacket(U8 bLength, U8 *abPayload)
{
        SEGMENT_VARIABLE(bCnt, U8, SEG_XDATA);
        SEGMENT_VARIABLE(bTxItState[8], U8, SEG_XDATA);

        abApi_Write[0] = CMD_CHANGE_STATE;// Use change state command
        abApi_Write[1] = 3;                             // Go to Ready mode
        bApi_SendCommand(2,abApi_Write);                // Send command to the radio IC
        vApi_WaitforCTS();                              // Wait for CTS

        bApi_WriteTxDataBuffer(bLength, abPayload);     // Write data to Tx FIFO
        vApi_WaitforCTS();                              // Wait for CTS

        // Set TX packet length
        abApi_Write[0] = CMD_SET_PROPERTY;              // Use property command
        abApi_Write[1] = PROP_PKT_GROUP;                // Select property group
        abApi_Write[2] = 1;                             // Number of properties to be written
        abApi_Write[3] = PROP_PKT_FIELD_1_LENGTH_7_0;     // Specify first property
        abApi_Write[4] = bLength;                       // Field length (variable packet length info)
        bApi_SendCommand(5,abApi_Write);                // Send command to the radio IC
        vApi_WaitforCTS();                              // Wait for CTS

        // Read IT
        // Read ITs, clear pending ones
        abApi_Write[0] = CMD_GET_INT_STATUS;            // Use interrupt status command
        abApi_Write[1] = 0;                             // Clear PH_CLR_PEND
        abApi_Write[2] = 0;                             // Clear MODEM_CLR_PEND
        abApi_Write[3] = 0;                             // Clear CHIP_CLR_PEND
        bApi_SendCommand(4,abApi_Write);                // Send command to the radio IC
        // Get the response
        bApi_GetResponse(8,bTxItState);                 // Make sure that CTS is ready

        // Start Tx
        abApi_Write[0] = CMD_START_TX;                  // Use Tx Start command
        abApi_Write[1] = 0;                             // Set channel number
        abApi_Write[2] = 0x30;                          // Ready state after Tx, start Tx immediately

        abApi_Write[3] = 0x00;                          // packet fields used, do not enter packet length here
        abApi_Write[4] = 0x00;                          // packet fields used, do not enter packet length here
        bApi_SendCommand(5,abApi_Write);                // Send command to the radio IC
        vApi_WaitforCTS();                              // Wait for CTS

        // Wait for packet sent interrupt
        while(EZRP_NIRQ == 1)

        // Read IT
        // Read ITs, clear pending ones
        abApi_Write[0] = CMD_GET_INT_STATUS;            // Use interrupt status command
        abApi_Write[1] = 0;                             // Clear PH_CLR_PEND
        abApi_Write[2] = 0;                             // Clear MODEM_CLR_PEND
        abApi_Write[3] = 0;                             // Clear CHIP_CLR_PEND
        bApi_SendCommand(4,abApi_Write);                // Send command to the radio IC
        bApi_GetResponse(8,bTxItState                   // Make sure that CTS is ready then get the response

        return 0;
}
```

3.7. Variable Packet Length (Sample Code 3)

Using variable packet lengths is very common in bidirectional communication. In Sample Code 3, the packet structure is set up in such a way that, after the preamble and sync word, the packet length info (1 byte) is included in the payload:





The radio can extract the length information from the received packet automatically. For this operation, it is desired to use a minimum of two fields. A fixed length field has to precede the variable length field, and the actual length information has to be the last byte of this fixed-length field. The variable field length operation for the Receive side shall be configured through these properties: PKT_LEN, PKT_LEN_FIELD_SOURCE, and PKT_LEN_ADJUST.

PKT_LEN determines which field is used with variable length and whether the length info will be put in the FIFO. PKT_OFFSET determines where in the field the length information starts. In the code example, the offset is set to zero. The first field is used to hold the length information, while the second field has variable length, and contains the actual data.

```
// General packet config (set bit order)
abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
abApi_Write[2] = 1;                         // Number of properties to be written
abApi_Write[3] = PROP_PKT_CONFIG1;          // Specify property
abApi_Write[4] = 0x80;                      // Separate RX and TX FIELD properties are used, payload data goes MSB first
bApi_SendCommand(5,abApi_Write);            // Send command to the radio IC
vApi_WaitforCTS();                          // Wait for CTS

// Set variable packet length
abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
abApi_Write[2] = 3;                         // Number of properties to be written
abApi_Write[3] = PROP_PKT_LEN;              // Specify property
abApi_Write[4] = 0x02;                      // Length is put in the Rx FIFO, FIELD 2 is used for the payload
abApi_Write[5] = 0x01;                      // FIELD 1 is used for the length information
abApi_Write[6] = 0x00;                      // No adjustment (FIELD 1 determines the actual payload length)
bApi_SendCommand(7,abApi_Write);            // Send command to the radio IC
vApi_WaitforCTS();                          // Wait for CTS

// Set packet fields for Tx (one field is used)
abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
abApi_Write[2] = 4;                         // Number of properties to be written
abApi_Write[3] = PROP_PKT_FIELD_1_LENGTH_12_8;          // Specify first property
abApi_Write[4] = 0x00;                      // Defined later (under bSendPacket() )
abApi_Write[5] = 0x00;                      // Defined later (under bSendPacket() )
abApi_Write[6] = 0x00;                      // No 4GFSK/whitening/Manchester coding for this field
abApi_Write[7] = 0xA2;                      // Start CRC calc. from this field, send CRC at the end of it
bApi_SendCommand(8,abApi_Write);            // Send command to the radio IC
vApi_WaitforCTS();                          // Wait for CTS

// Set packet fields for Rx (two fields are used)
//FIELD1 is fix 1byte length, used for PKT_LEN
abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
abApi_Write[2] = 4;                         // Number of properties to be written
abApi_Write[3] = PROP_PKT_RX_FIELD_1_LENGTH_12_8;       // Specify first property
abApi_Write[4] = 0x00;                      // Max. field length (variable packet length)
abApi_Write[5] = 0x01;                      // Max. field length (variable packet length)
abApi_Write[6] = 0x00;                      // No 4GFSK/whitening/Manchester coding for this field
abApi_Write[7] = 0x82;                      // Start CRC calc. from this field, enable CRC calc.
bApi_SendCommand(8,abApi_Write);            // Send command to the radio IC
//FIELD2 is variable length, contains the actual payload
vApi_WaitforCTS();                          // Wait for CTS
abApi_Write[0] = CMD_SET_PROPERTY;          // Use property command
abApi_Write[1] = PROP_PKT_GROUP;            // Select property group
abApi_Write[2] = 4;                         // Number of properties to be written
abApi_Write[3] = PROP_PKT_RX_FIELD_2_LENGTH_12_8;       // Specify first property
abApi_Write[4] = 0x00;                      // 1 byte (containing variable packet length info)
abApi_Write[5] = MAX_RX_LENGTH;             // 1 byte (containing variable packet length info)
abApi_Write[6] = 0x00;                      // No 4GFSK/whitening/Manchester coding for this field
abApi_Write[7] = 0x0A;                      // Start CRC calc. from this field, check CRC at the end, enable CRC calc.
bApi_SendCommand(8,abApi_Write);            // Send command to the radio IC
vApi_WaitforCTS();
```

Packet reception is implemented exactly as shown in the Sample Code 2; the only exception being that an acknowledgement packet is transmitted upon valid packet reception.

```
// Wait for an interrupt
if(EZRP_NIRQ == 0)
{
        // Read ITs, clear pending ones
        abApi_Write[0] = CMD_GET_INT_STATUS;        // Use interrupt status command
        abApi_Write[1] = 0;                         // Clear PH_CLR_PEND
        abApi_Write[2] = 0;                         // Clear MODEM_CLR_PEND
        abApi_Write[3] = 0;                         // Clear CHIP_CLR_PEND
        bApi_SendCommand(4,abApi_Write);            // Send command to the radio IC
        bApi_GetResponse(8,bItStatus);              // Get the response

        //check the reason for the IT
        if((bItStatus[2] & 0x08) == 0x08)
        {//CRC error
                SetLed(1);                          // Show LED combination for CRC error
                SetLed(4);
        }
        if((bItStatus[2] & 0x10) == 0x10)
        {//packet received
                // Get RSSI
                bApi_GetFastResponseRegister(FAST_RESPONSE_REG_C,1,&bRssi);// Get length of the packet
                abApi_Write[0] = CMD_FIFO_INFO;     // Use FIFO info command to get the length
                bApi_SendCommand(1,abApi_Write);    // Send command to the radio IC
                bApi_GetResponse(2,&abApi_Read[0]); // Get the response
                // Read out the payload after the length
                bApi_ReadRxDataBuffer(abApi_Read[0],&abApi_Read[2]);

                // Check the packet content: figure out what packet is received (data or ack)
                if ((abApi_Read[2]=='B') && (abApi_Read[3]=='U') && (abApi_Read[4]=='T') &&
(abApi_Read[5]=='T') && (abApi_Read[6]=='O') && (abApi_Read[7]=='N'))// Data packet received
                {
                        SetLed(2);                  // Turn on the appropriate LED
                        for(wDelay=0; wDelay<30000; wDelay++);      // Wait to show LED
                        ClearLed(2);                // Turn off the corresponding LED
                        SetLed(1);                  // Turn on the appropriate LED

                        // Put the payload to Tx FIFO
                        abPayload[0] = LENGTH_OF_ACK_PAYLOAD-1;  // write the length of payload
                        abPayload[1] = 0x41;        // write 0x41 ('A')
                        abPayload[2] = 0x43;        // write 0x43 ('C')
                        abPayload[3] = 0x4B;        // write 0x4B ('K')

                        bSendPacket(LENGTH_OF_ACK_PAYLOAD, abPayload);    // Send ACK packet

                        for(wDelay=0; wDelay<30000; wDelay++);      // Wait to show LED
                        ClearLed(1);                // Turn off the corresponding LED
                }
```

3.8. Using 4(G)FSK Modulation (Sample Code 4)

Si446x radios can be configured to transmit packets with 4(G)FSK modulation. In such a case, one symbol represents two bits; this is in contrast to 2(G)FSK, where one symbol represents one bit. As a result, the bit rate can be doubled with the same symbol rate. Sample Code 4 demonstrates how to configure the radio IC using 4(G)FSK modulation.

In order for the device to operate in 4(G)FSK mode, the following must be accomplished:

- Generate a new modem_params.h under WDS, where 4(G)FSK modulation is enabled; this will contain the correct parameters for the modem.
- In PKT_CONFIG1 property, enable 4(G)FSK; this setting is for the packet handler.
- Use a FIELD (see "3.5. Using the FIELDs (Sample Code 2)" on page 32.), and enable 4(G)FSK modulation in PKT_FIELD_x_CONFIG.
- Configure PKT_CHIP_MAP and MODEM_FSK4_MAP properties to achieve the desired symbol map (see Table 3, “Settings for the 24 Possible Symbol Maps,” on page 46).

Relevant segments of the sample code are shown below.

```
// General packet config (set bit order)
abApi_Write[0] = CMD_SET_PROPERTY;           // Use property command
abApi_Write[1] = PROP_PKT_GROUP;             // Select property group
abApi_Write[2] = 2;                          // Number of properties to be written
abApi_Write[3] = PROP_PKT_CONFIG1;           // Specify property
abApi_Write[4] = 0x20;                       // 4FSK enabled, payload data goes MSB first (PKT_CONFIG1)
abApi_Write[5] = PH_4FSK_MAP;                // Packet handler 4(G)FSK map (PKT_CHIP_MAP)
bApi_SendCommand(6,abApi_Write);     // Send command to the radio IC
vApi_WaitforCTS();                           // Wait for CTS

// Set packet fields (use only one field out of the available five)
abApi_Write[0] = CMD_SET_PROPERTY;           // Use property command
abApi_Write[1] = PROP_PKT_GROUP;             // Select property group
abApi_Write[2] = 4;                          // Number of properties to be written
abApi_Write[3] = PROP_PKT_FIELD_1_LENGTH_12_8;  // Specify first property
abApi_Write[4] = 0x00;
abApi_Write[5] = 0x08;           // 8 byte long packet field
abApi_Write[6] = 0x10;           // 4(G)FSK enabled, no whitening/Manchester coding for this field
abApi_Write[7] = 0x8A;      // PKT_FIELD_1_CRC_CONFIG   Start CRC calc. from this field, check CRC at the end
bApi_SendCommand(8,abApi_Write);       // Send command to the radio IC
vApi_WaitforCTS();                           // Wait for CTS

// Set 4(G)FSK map
abApi_Write[0] = CMD_SET_PROPERTY;           // Use property command
abApi_Write[1] = PROP_MODEM_GROUP;        // Select property group
abApi_Write[2] = 1;                          // Number of properties to be written
abApi_Write[3] = PROP_MODEM_FSK4_MAP;  // Specify property
abApi_Write[4] = MODEM_4FSK_MAP;           // Modem 4(G)FSK map (MODEM_FSK4_MAP)
bApi_SendCommand(5,abApi_Write);             // Send command to the radio IC
vApi_WaitforCTS();                           // Wait for CTS
```

As shown in Table 3, there are 24 possible permutations of how to map pairs of data bits into modulation level 1–4 (i.e., into the four frequency channels). Table 3 summarizes all variations, where “–3”, “–1”, “+1”, and “+3” refer to the four modulation levels: “–3” refers to <center frequency – 3 x deviation>, “–1” refers to <center frequency – 1 x deviation>, etc.

Table 3. Settings for the 24 Possible Symbol Maps

No.	4(G)FSK				TX/RX	TX	RX
	4-LEVEL CODE				PKT_CHIP_MAP	MODEM_FSK4_MAP	MODEM_FSK4_MAP
	–3	–1	+1	+3	0x1207	0x203F	0x203F
1	`00	`01	`11	`10	1E	B4	B4
2	`00	`01	`10	`11	1B	E4	E4
3	`00	`11	`01	`10	36	78	9C
4	`00	`11	`10	`01	39	6C	6C
5	`00	`10	`01	`11	27	D8	D8
6	`00	`10	`11	`01	2D	9C	78
7	`01	`00	`11	`10	4E	B1	B1
8	`01	`00	`10	`11	4B	E1	E1
9	`01	`11	`00	`10	72	72	8D
10	`01	`11	`10	`00	78	63	2D
11	`01	`10	`00	`11	63	D2	C9
12	`01	`10	`11	`00	6C	93	39
13	`11	`00	`01	`10	C6	39	93
14	`11	`00	`10	`01	C9	2D	63
15	`11	`01	`00	`10	D2	36	87
16	`11	`01	`10	`00	D8	27	27
17	`11	`10	`00	`01	E1	1E	4B
18	`11	`10	`01	`00	E4	1B	1B
19	`10	`00	`01	`11	87	C9	D2
20	`10	`00	`11	`01	8D	8D	72
21	`10	`01	`00	`11	93	C6	C6
22	`10	`01	`11	`00	9C	87	36
23	`10	`11	`00	`01	B1	4E	4E
24	`10	`11	`01	`00	B4	4B	1E

Note that certain symbol maps require different settings on the TX side of the link than on the RX side of the link (e.g., symbol map No. 1 vs No. 3). This must be taken into account in case of bidirectional communication, as MODEM_FSK4_MAP may need to be updated at every TX-RX / RX-TX state change, and this “extra” SPI communication takes some time. The recommended preamble length in case of 4(G)FSK modulation is min. 48 bits if AFC is ON, 40 bits if AFC is OFF, while the preamble detection threshold should be at least 16 bits.

NOTES:

Contact Information

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page:
https://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.