
PROGRAMMING GUIDE FOR EZRADIOPRO® Si4X6X DEVICES

1. Introduction

This document is intended to serve as a guide for application development with EZRadioPRO® radio ICs. It introduces the major parts of the hardware platform, such as the RF Pico board, which contains the radio and the necessary RF components required to operate the device according to a desired regulatory standard. It also introduces the 8-bit wireless motherboard (WMB), which is required to control the radio, evaluate the RF parameters, and develop custom application programs. Besides the hardware, it also describes the application programming interface (API) that makes it possible for the WMB and RF pico board to communicate with each other. Using the software tools provided by Silicon Labs and following this programming guide will make software development as easy as possible, as these items will assist you in configuring the radio effectively. Additionally, the first boot of the radio and the whole configuration process are clearly described so that software developers can primarily concentrate on their own applications without experiencing time-consuming configuration problems. Several example projects are also provided as good starting points for real applications. A layered software approach is followed in all the source codes. The software modules are logically separated, and they focus on their own specific tasks. The document refers to the corresponding data sheets, manuals, and application notes.

2. Supported Radio Types

This document provides programming guidance for the following EZRadioPRO RF ICs:

- Si4060 Transmitter
- Si4063 Transmitter
- Si4362 Receiver
- Si4438 Transceiver
- Si4460 Transceiver
- Si4461 Transceiver
- Si4463 Transceiver
- Si4464 Transceiver
- Si4467 Transceiver
- Si4468 Transceiver

AN633

3. Development Kits

The EZRadioPRO development kits contain two complete RF nodes of different radio ICs. See Table 1.

Table 1. EZRadioPRO Development Kit Content

Description	Part Number				
	4060-868-PDK	4063-915-PDK	4461-868-PDK	4438-490-PDK	4463-915-PDK
RF Pico Board	4060-PCE10B868 (1 pc) 4362-PRXB868 (1 pc)	4063-PCE20B915 (1 pc) 4362-PRXB915 (1 pc)	4461-PCE14D868 (2 pcs)	4438-PCE20D490 (2pcs)	4463-PCE20C915 (2pcs)
Mother-board	2 pcs MSC-WMB93X				
MCU Pico Board	2 pcs UPPI-930-RF				
Antenna	868 MHz	915 MHz	868 MHz	490 MHz	915 MHz
Others	2 pcs USB Cable				
	Kit user's guide				

4. The Wireless Motherboard Hardware Platform

The wireless motherboard platform is a demo, evaluation, and development platform for EZRadioPRO radio ICs. It consists of a wireless motherboard and interchangeable MCU and RF pico boards.

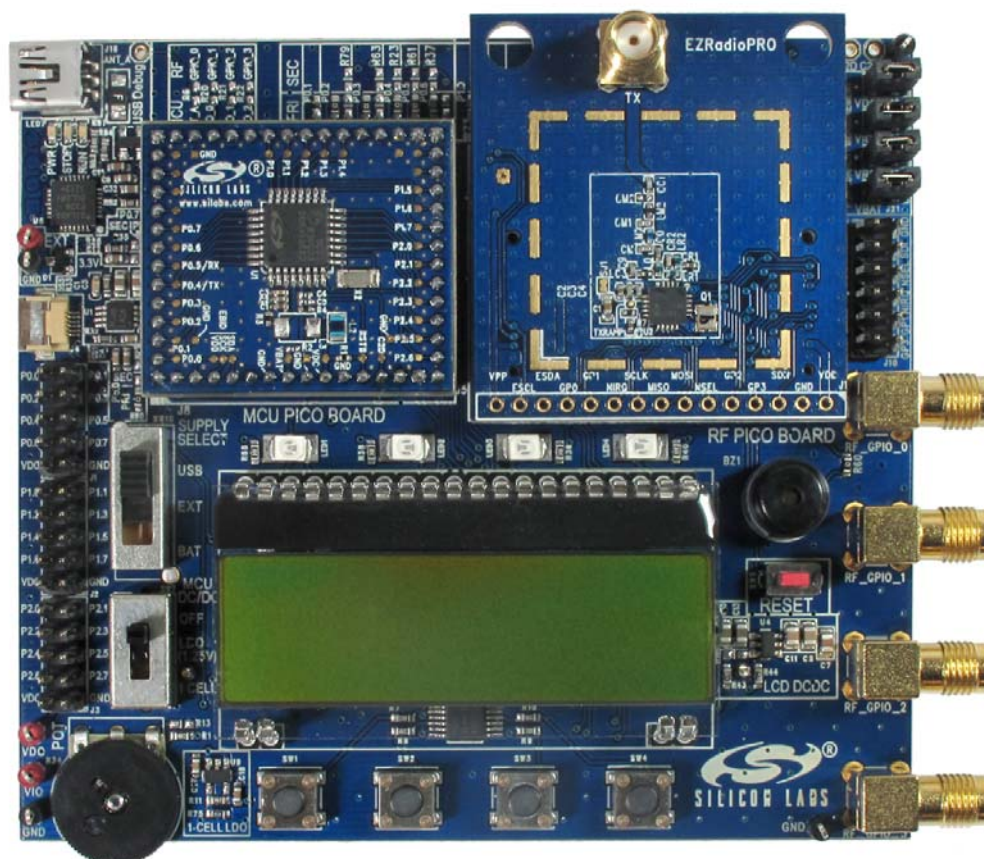


Figure 1. 8-bit Wireless Motherboard Platform

4.1. The Wireless Motherboard

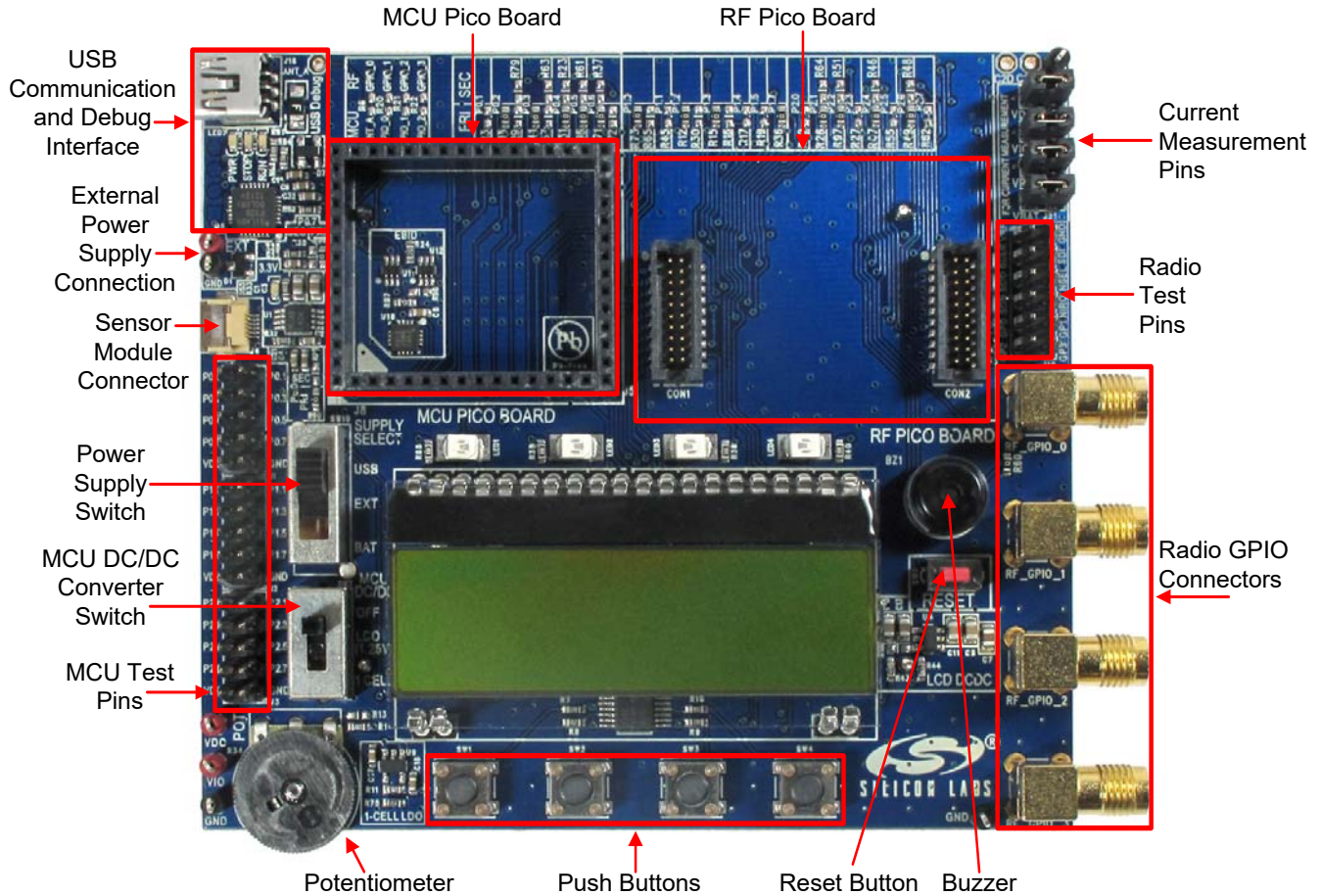


Figure 2. Wireless Motherboard

The wireless motherboard contains four pushbuttons, four LEDs, and a buzzer as simple user interfaces. A graphical LCD displays menu items for range testing purposes and a potentiometer demonstrates analog capabilities of the MCU. A switch supports the power options of the MCU's built-in dc/dc converter. Using the current measurement jumpers, current consumption can be measured separately either for the MCU, the radio, or the peripherals. The motherboard contains test pins for all I/O pins of the MCU and for all digital pins of the radio. In addition, there are SMA connectors for the GPIOs of the radio for test equipment connection. A USB communication interface as well as a built-in Silicon Labs USB-to-C2 debug adapter are integrated onto the board so that the wireless motherboard (WMB) can be directly connected via USB to the PC for downloading and debugging code on the MCU.

An interface connection towards sensor modules can also be found. The MCU is also connected to the RF pico board through a connector pair.

4.2. Power Scheme

The power source of the platform can be selected with the power supply selector switch “SUPPLY SELECT” on the WMB board. If this switch is in the “USB” position, supply voltage is provided by the PC that is connected to the “J16” mini USB connector. If this switch is in the “BAT” position, the supply voltage is provided by two AA batteries in the battery holder on the bottom side of the board. If the “SUPPLY SELECT” switch is in the “EXT” position, supply voltage is provided by an external power source through the “TP7” and “TP9” points.

Using the “MCU dc/dc” switch, the internal dc/dc converter of the C88051F930 MCU on the MCU pico board can be activated if the connected pico board supports this function. If the switch is in the “OFF” position, the MCU's dc/dc converter is inactive and the supply voltage is only determined by the state of the “SUPPLY SELECT” switch.

Positioning the switch to either the “LDO (1.25 V)” or “1 CELL” position will turn on the MCU's dc/dc converter by connecting 1.25–1.5 V supply voltage to the VBAT pin and removing external power from the VDC pin. The MCU will provide 1.9 V in default setting on its VDC pin to all the other connected loads. Since this current is limited, it may be necessary to disconnect or disable some loading part of the board. For further details, see the MCU data sheet and the board schematic. The board schematic can be found in the EZRadioPRO Development Kit User's Guide. A complete CAD design pack of the board is also available at www.silabs.com.

4.3. RF Pico Board

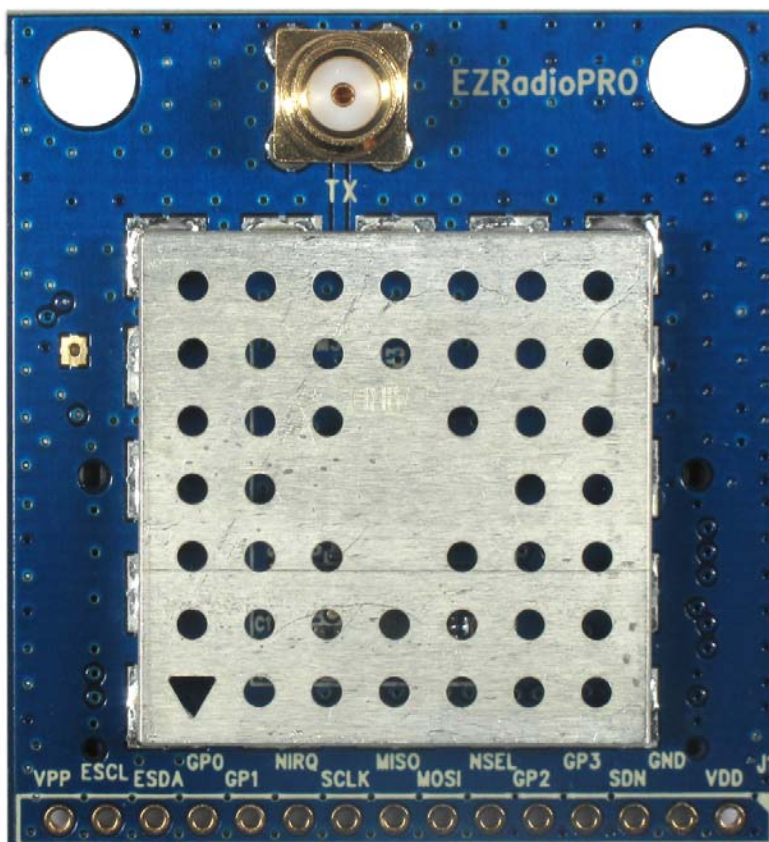


Figure 3. RF Pico Board Front Side

The RF pico board is a radio module that contains an EZRadioPRO radio IC, matching network and an SMA connector on the top side. These components apart from the antenna connector are covered by a metal shield for noise reduction. The digital signals of the radio (SCLK, SDI, SDO, NSEL, SCL, SDA, VDD and GND) can be accessed on test points at the edge of the board. The boards also have a factory loaded board identification memory (EBID) on the bottom side that contains data that describes the board properties. Via the unified RF pico connector pair on the bottom side of the board, any RF pico board can be connected to the WMB.

Table 2. Connections between the WMB Board and the RF Pico Board

Si446x, Si4362, Si406x, Si4438			WMB		C8051F930
Pin Number	Pin Name	Pin Function	RF Pico board J1 connector	WMB Con2 connector	Pin Name
EP,18	GND	Ground	2	1,2,19,20	GND
8	VDD	Voltage Supply input	1	17,18	VDD
11	NIRQ	Interrupt output active low	10	7	P0.1
1	SDN	Shutdown input active high	3	8	P2.3
15	NSEL	SPI select input	6	6	P1.4
12	SCLK	SPI clock input	9	5	P1.0
14	SDI	SPI data input	7	3	P1.2
13	SDO	SPI data output	8	4	P1.1
9	GPIO_0	General Purpose I/O	12	11	P2.6 (2nd)
10	GPIO_1	General Purpose I/O	11	12	P1.3
19	GPIO_2	General Purpose I/O	5	13	P2.5
20	GPIO_3	General Purpose I/O	4	14	P2.4

A schematic of an RF Pico Board can be found in the EZRadioPRO Development Kit User's Guide. A complete CAD design pack of all boards is also available at www.silabs.com.

4.4. Setting up and Connecting the WMB to the PC

Steps for connecting the platform to the PC:

1. Connect an RF Pico Board to the WMB board through the CON1 and CON2 connectors.
2. Insert a UPPI-930-RF MCU pico board in the connectors J5, J6, J7, J8 on the WMB. The dotted corner of the C8051F930 MCU has to point to the triangle symbol on the WMB.
3. Connect an antenna to the SMA connector on the RF Pico Board.
4. Select the desired power source with the SUPPLY SELECT switch.
5. Ensure that all the CURRENT MEASUREMENT jumpers are in place.
6. Connect the WMB board to a USB port of the PC.
7. Wait for Windows to install the driver of the debug interface if necessary.

5. Software Development Tools

5.1. Wireless Development Suite

Silicon Labs provides two software tools to help with EZRadioPRO software development: the wireless development suite (WDS) and the Silicon Labs integrated development environment (IDE). Both software tools are available at www.silabs.com. The recommended starting point for Si406x, Si4362, Si446x, and Si4438 development is the WDS software tool. After connecting one of the hardware platforms to the PC, WDS is able to identify the connected boards by reading the EBID memories of the boards. The EZConfigPRO Setup GUI is part of the WDS program. This setup interface provides an easy path to quickly selecting and loading the desired configuration for the Si406x, Si4362, Si446x, and Si4438 device. The EZConfigPRO Setup allows four different methods for device setup. After the desired configuration is selected, the program gives the option to configure directly the EZRadioPRO chip of the connected hardware, or to modify a selected example code with the configuration and download it to the connected hardware. It is possible to export and save the example projects and radio configuration file (radio_config.h) from the WDS. Using the header file generated by the WDS is highly recommended. Manual editing in the header file may cause problems and prevent the radio from working correctly. For more complete information on WDS and EZConfigPRO usage, refer to the WDS User's Guide. Figure 4 is a summary of the WDS configuration workflow.

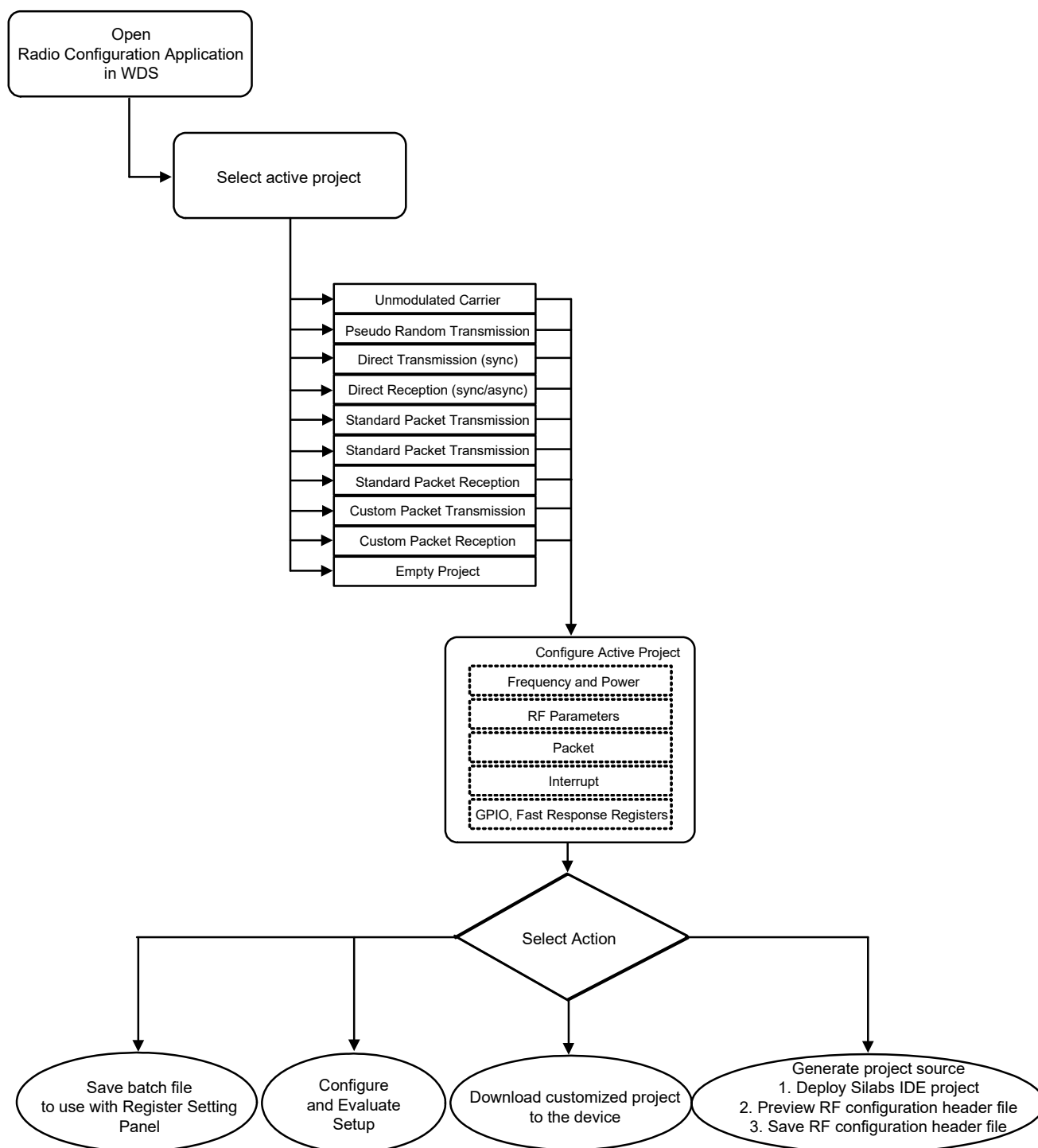


Figure 4. Device Configuration Options

For more details about the selectable actions, refer to the WDS User Guide for EZRadioPRO devices.

5.2. Silicon Labs IDE

The Silicon Laboratories integrated development environment (IDE) is a standard tool for program development for any Silicon Labs 8-bit MCUs, including the C8051F930 that is used on the hardware platforms described in this document. The Silicon Laboratories IDE integrates a project manager, a source-code editor, source-level debugger, and an in-system flash programmer. The IDE interfaces to third party development tool chains to provide system designers a complete embedded software development environment. The Keil Demonstration Toolset includes a compiler, linker, and assembler and easily integrates into the IDE.

Workflow for downloading and running a project:

1. Connect the hardware platform to the PC according to the description of the used platform.
2. Start Silicon Labs IDE (IDE 4.40 or higher required) on your computer.
3. Select **Project**→**Open Project...** to open a previously saved project.
4. Before connecting to the target device, several connection options may need to be set. Open the Connection Options window by selecting **Options**→**Connection Options...** in the IDE menu.
5. Select USB Debug Adapter in the "Serial Adapter" section.
6. If more than one adapter is connected, choose the appropriate serial number from the drop-down list.
7. Check "Power target after disconnect" if the target board is currently being powered by the USB Debug Adapter. The board will remain powered after a software disconnect by the IDE.
8. Next, the correct "Debug Interface" must be selected. Check the C2 Debug Interface.
9. Once all the selections are made, click the **OK** button to close the window.
10. Click the **Connect** button in the toolbar or select **Debug**→**Connect** from the menu to connect to the MCU of the platform.
11. Erase the flash of the MCU in the **Debug**→**Download object code**→**Erase** all code space menu item.
12. Download the desired HEX file either by hitting the Download code (Alt+D) toolbar button or from the **Debug**→**Download** object code menu item.
13. Hit the **Disconnect** toolbar button or invoke the **Debug**→**Disconnect** menu item to release the device from halt and to let it run.

5.3. Toolstick Terminal

The ToolStick Terminal program provides the standard terminal interface to the target microcontroller's UART. However, instead of requiring the usual RS-232 and COM port connection, ToolStick Terminal uses the USB interface of the ToolStick Base Adapter to provide the same functionality. The firmware on the target microcontroller does not need to be customized to use the UART and communicate with ToolStick Terminal. The firmware on the microcontroller should write to the UART as it would in any standard application and all of the translation is handled by the ToolStick Base Adapter. The ToolStick Base Adapter is integrated on the WMB and is also part of the RFStick platform as a separate device.

The ToolStick Terminal program is part of the Silicon Labs IDE and is also available as a separate application. Both can be installed as part of the Silicon Labs 8-bit Microcontroller Studio from <http://www.silabs.com/products/mcu/Pages/8-bit-microcontroller-software.aspx>.

The IDE and its built-in Toolstick Terminal can communicate with the target MCU simultaneously on the C2 interface and on the UART respectively.

To use the ToolStick Terminal in the IDE (above v4.60.00) follow these steps:

1. Open the Silabs IDE from the **Start**→**Programs**→**Silicon Laboratories** menu.
2. Go to the **Options** → **Connection Options** menu and select the desired ToolStick Base Adapter from the drop down list.
3. Click on the **Connect** button to connect the IDE to the target MCU via the C2 interface.
4. From the Tools menu start the ToolStick Terminal. In the top left-hand corner of the Terminal application, go to the **ToolStick** → **Settings** menu and set the communications parameters. Now the ToolStick Terminal is ready for use. In the Receive Data window, text indicating the received characters will appear.

In addition to the standard two UART pins (TX and RX), there are two GPIO/UART handshaking pins on the ToolStick Base Adapter. On both the WMB and RFStick platforms GPIO0 is used for the internal purpose of the WDS to select between the C2 interface of the target MCU and the EBID MCU. GPIO1 is not connected. Although the separate ToolStick Terminal application provides the functionality to control these GPIOs, default settings for GPIO0 should not be changed.

5.4. Prerequisites for Code Development

All the sample projects have a unified structure and common driver set. This section provides a brief introduction of the structure of the example software projects. The settings in the sample project files assume that some Silicon Labs or third party software tools are already installed on the PC where the sample project is going to be compiled. The tools that need to be installed depends on the functionality to be used. The following list contains a complete set of such programs:

- Silicon Laboratories IDE—Used to open the preconfigured project files and manage the build process.
- Keil C51 v9.0+ or SDCC v3.0+—Compilers to use with the Silicon Laboratories IDE to manage build process.
- Silicon Labs Flash Programming Utility (optional)—Needed only if programming outside the Silicon Labs IDE is necessary.
- Make (optional)—This tool is needed in case another compiler is used or the build process takes place outside of the SiLabs IDE. "Makefile" can be generated with the `wsp2make.exe` utility. It is only recommended for advanced users since it may need manual editing.

5.5. Supported Compilers

The projects come with one Silicon Labs IDE project file which is prepared to use the Keil C51 toolchain. An evaluation version of the Keil toolchain can be downloaded from the Keil website at <http://www.keil.com>. The Keil free evaluation version can be unlocked to become a full version with no code placement limitation. Visit the Silicon Labs website at <http://www.silabs.com/products/mcu/Pages/8-bit-microcontroller-software.aspx#keil-pk51> to get the full license. The project files in examples assume that the Keil tool chain is installed to the C:\Keil directory. The location of the Keil tool chain can be easily changed in the Silabs IDE in the Project Tool Chain Integration menu. However, the sample projects can be compiled not only with the two mentioned compilers, but with almost any ANSI C compiler for 8051 architecture with little or no modifications. Each project already contains a "Makefile" in order to provide an easy and convenient way to compile the code outside the Silicon Labs IDE with the tool chain of choice. Each sample project described in this document contains a compiled version of the source code in Intel hex format that is widely supported by a variety of programming and debugging tools. The compiled file in the projects has been generated using the Silicon Labs IDE and the Keil C51 tool chain. AN104: Integrating Keil 8051 Tools into the Silicon Labs IDE covers toolchain integration and license management in more detail.

6. Radio Hardware Interface

The EZRadioPRO devices can be controlled by the host MCU over an SPI bus and six additional signals. The user has access to the radio's API via the SPI bus.

Table 3. Serial Peripheral Interface Signals

Signal	Description
SCLK	serial clock output from master
SDI	master output, slave input
SDO	master input, slave output
NSEL	slave select, active low

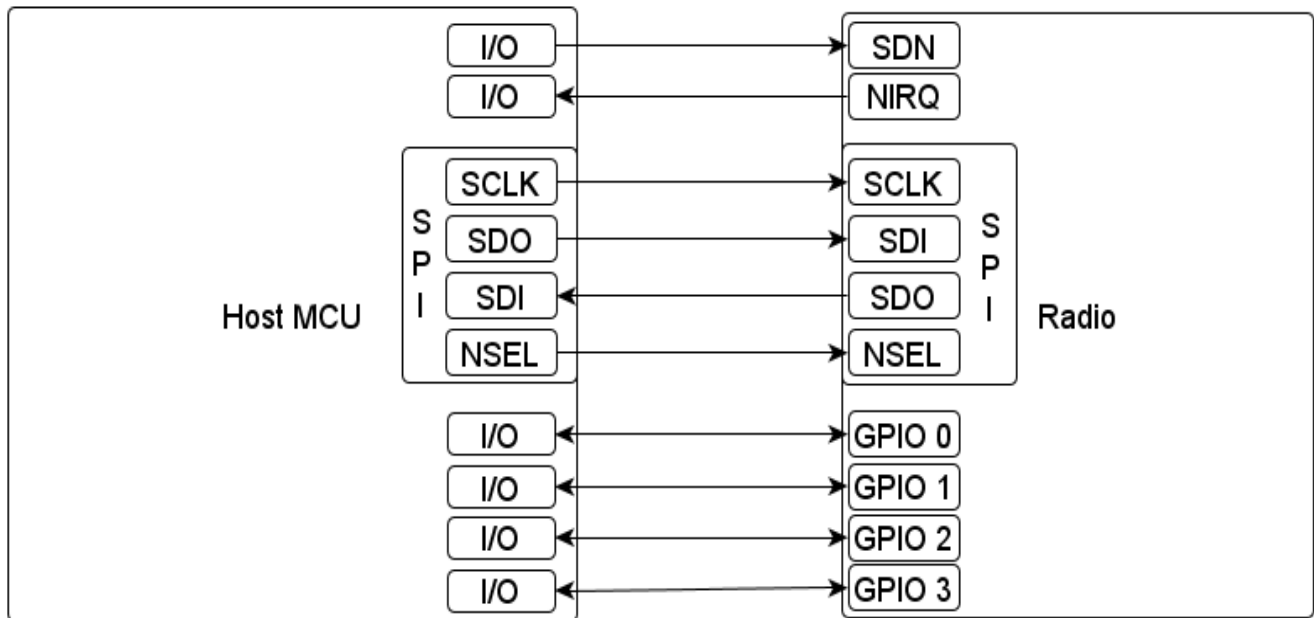


Figure 5. Connections between the Host MCU and the Radio

The high state of the shutdown (SDN) pin is used to completely disable the radio and put the device into the lowest power consumption state. 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.

7. Application Programming Interface

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 API commands are listed in Table 4. The following sections describe the SPI transactions of sending commands and getting information from the chip.

Table 4. Command Summary

Command ID	Name	Description
BOOT_COMMANDS		
0x02	POWER_UP	Command to power-up the device and select the operational mode and functionality.
COMMON_COMMANDS		
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 one or more properties.
0x12	GET_PROPERTY	Retrieves the value of one or more properties.
0x13	GPIO_PIN_CFG	Configures the GPIO pins.
0x15	FIFO_INFO	Access the current byte counts in the TX and RX FIFOs and provide for resetting the FIFOs.
0x20	GET_INT_STATUS	Returns the interrupt status of ALL the possible interrupt events (both STATUS and PENDING). Optionally, it may be used to clear latched (PENDING) interrupt events.
0x33	REQUEST_DEVICE_STATE	Request current device state and channel.
0x34	CHANGE_STATE	Manually switch the chip to a desired operating state.
0x38	OFFLINE_RECAL	Recalibrates due to temperature change.
0x44	READ_CMD_BUFF	Used to read CTS and the command response.
0x50	FRR_A_READ	Reads the fast response registers (FRR) starting with FRR_A.
0x51	FRR_B_READ	Reads the fast response registers (FRR) starting with FRR_B.
0x53	FRR_C_READ	Reads the fast response registers (FRR) starting with FRR_C.
0x57	FRR_D_READ	Reads the fast response registers (FRR) starting with FRR_D.

Table 4. Command Summary (Continued)

Command ID	Name	Description
IR_CAL_COMMANDS		
0x17	IRCAL	Image rejection calibration.
0x19	IRCAL_MANUAL	Image rejection calibration.
TX_COMMANDS		
0x31	START_TX	Switches to TX state and starts transmission of a packet.
0x37	TX_HOP	Hop to a new frequency while in TX.
0x66	WRITE_TX_FIFO	Writes data byte(s) to the TX FIFO.
RX_COMMANDS		
0x16	PACKET_INFO	Returns information about the length of the variable field in the last packet received and (optionally) overrides field length.
0x22	GET_MODEM_STATUS	Returns the interrupt status of the Modem Interrupt Group (both STATUS and PENDING). Optionally, it may be used to clear latched (PENDING) interrupt events.
0x32	START_RX	Switches to RX state and starts reception of a packet.
0x36	RX_HOP	Manually hop to a new frequency while in RX mode.
0x77	READ_RX_FIFO	Reads data byte(s) from the RX FIFO.
ADVANCED_COMMANDS		
0x14	GET_ADC_READING	Performs conversions using the Auxiliary ADC and returns the results of those conversions.
0x21	GET_PH_STATUS	Returns the interrupt status of the Packet Handler Interrupt Group (both STATUS and PENDING). Optionally, it may be used to clear latched (PENDING) interrupt events.
0x23	GET_CHIP_STATUS	Returns the interrupt status of the Chip Interrupt Group (both STATUS and PENDING). Optionally, it may be used to clear latched (PENDING) interrupt events.

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

7.1. Sending Command to Radio

The behavior of the radio can be changed by sending API 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. For most of the commands the host MCU does not expect any response from the radio chip. Other commands are used to read back a property from the chip such as checking the interrupt status flags, reading the transmit/receive FIFOs.

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 clear

to send (CTS) signal shows the actual status of the command buffer of the radio. It can be monitored over the SPI or on GPIOs, or the chip can generate an interrupt if it is ready to receive the next command. These three options are detailed below.

7.2. Checking the Radio is Ready to Receive Command

7.2.1. Software Polling Method

To ensure 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 completed, the NSEL should be pulled back to high. If the CTS byte is 0xFF, it means that the radio processed the last 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.

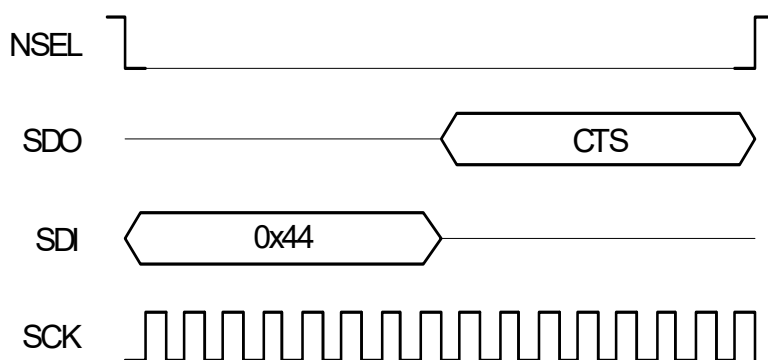


Figure 6. Polling the Radio Availability

7.2.2. GPIO Checking Method

Any GPIO can be configured for monitoring the CTS. GPIOs can be configured to go either high or low when the chip has completed 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.

7.2.3. NIRQ Interrupt Checking Method

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. 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 to communicate this error to the MCU by the 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.

All the commands that are sent to the radio 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.

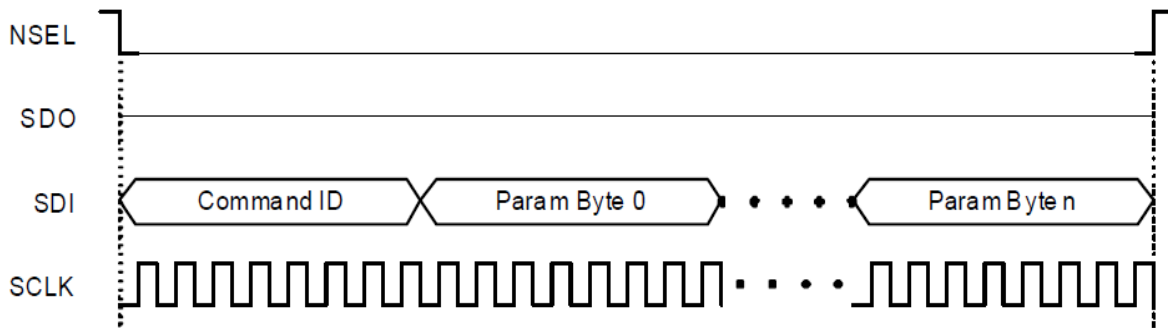


Figure 7. Host MCU Sends Command to Radio

7.3. Getting Response to a Command from the Radio

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.

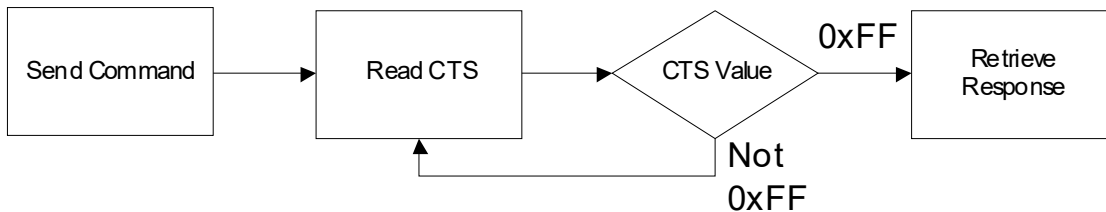


Figure 8. Read Procedure

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.

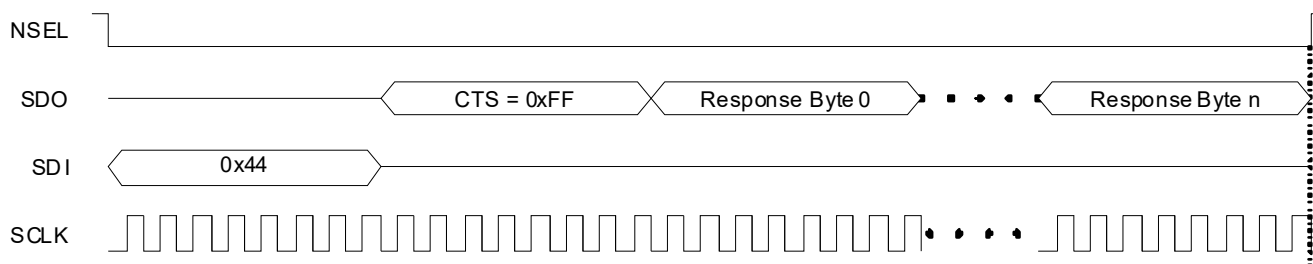


Figure 9. Read the Response from Radio

If the CTS is polled over the SPI bus, first the host MCU should pull the NSEL pin low. This action 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 clock cycles on the SCLK pin, as many 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.

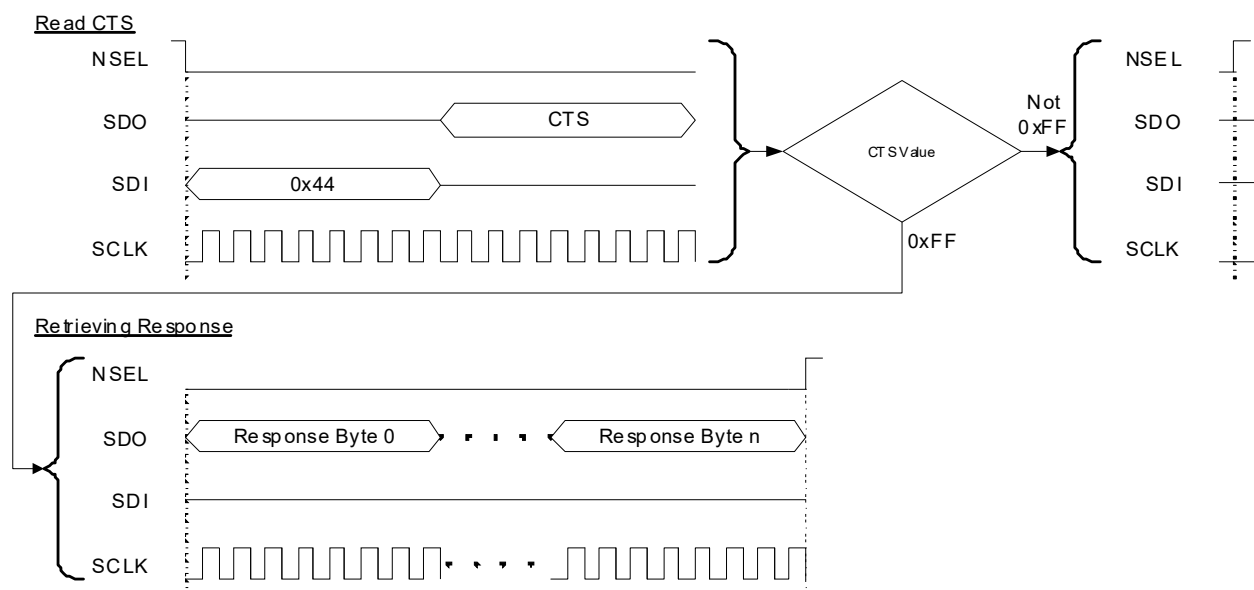


Figure 10. Monitor CTS and Read the Response on the SPI Bus

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:

- 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.
- 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.
- 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.

7.4. Using Fast Response Registers

There are several types of status information that can be read out from the radio faster. 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.

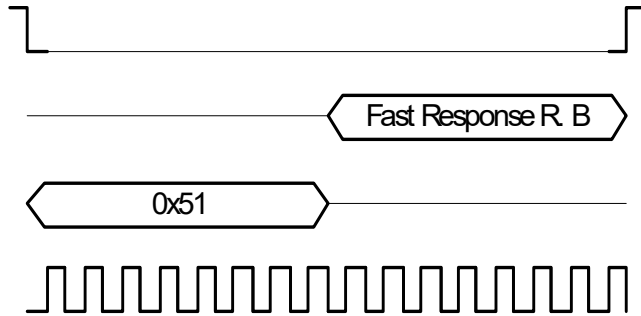


Figure 11. Reading a Single Fast Response Register

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.

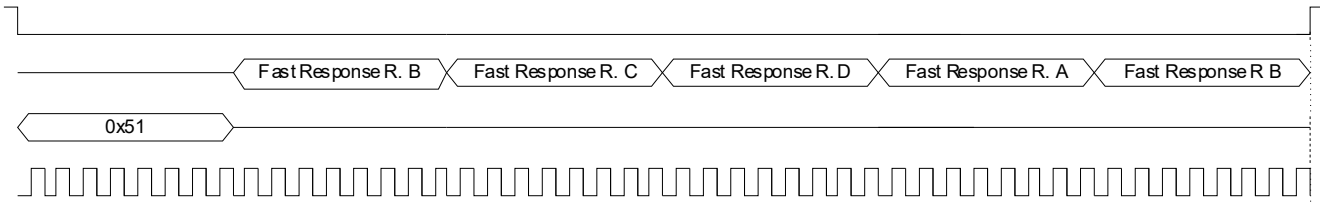


Figure 12. Reading More Fast Response Registers in a Single SPI Transaction

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.

7.5. Write and Read the FIFOs

There are two 64-byte FIFOs for RX and TX data in the Si4x6x.

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.

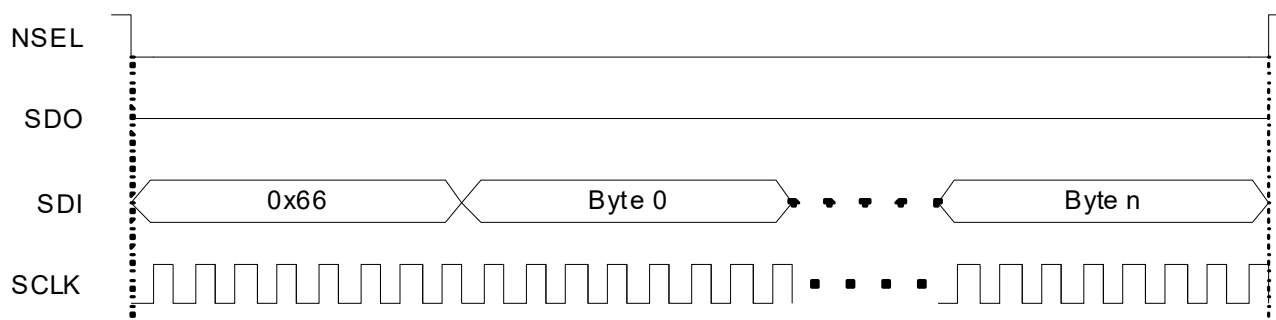


Figure 13. Transmit FIFO Write

If the host MCU needs 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.

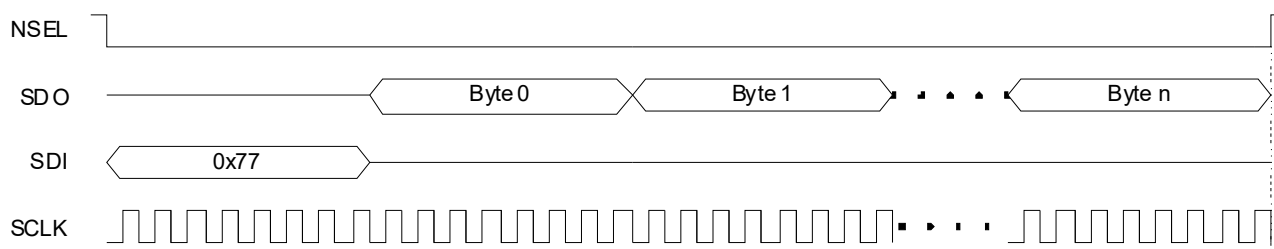


Figure 14. Receive FIFO Read

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, issuing a GET_CHIP_STATUS or GET_INTERRUPT_STATUS command, to clear the pending interrupt and release the NIRQ pin.

7.6. SPI Communication Capture Example

Figure 15 shows an actual SPI communication capture taken by a logic analyzer. The signals being monitored are SDI, SDO, and nSEL between the radio IC and the host MCU. The first command being sent is FIFO_INFO (command ID 0x15) with an input parameter of 0x01, which will reset the TX FIFO. Right after sending the FIFO_INFO command, the CTS is being monitored (0x44). For the first attempt, it is still busy; the returned value is NOT 0xFF, so nSEL goes back high. For the second attempt, the CTS value will be 0xFF, meaning that the radio IC has processed the command (i.e. resetting the TX_FIFO), and it is ready to provide the response bytes of the FIFO_INFO command. Therefore, nSEL stays low, and two dummy bytes are provided via SDI to read out the two response bytes through SDO. The response bytes are RX FIFO count and TX FIFO count numbers for the FIFO_INFO command. The next sequence is sending GET_STATUS command with three input bytes, all zeroes, to clear all ITs, and then retrieving the response bytes of the command after checking CTS. Once the ITs are cleared, six bytes are being written to the TX FIFO via WRITE_TX_FIFO command (0x66). Lastly, START_TX command is being sent to initiate an actual transmission. For more details of the commands described here, please see the html-based API documentation.

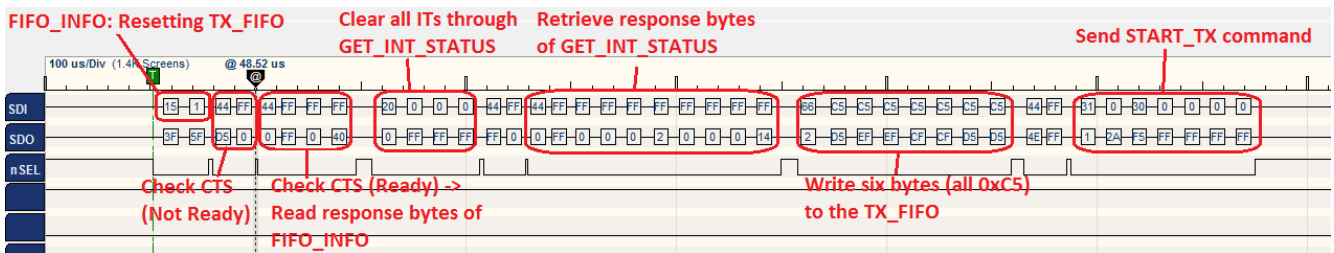


Figure 15. SPI Communications Example

8. Radio Initialization

8.1. State Transitions of the EZRadioPRO Devices

Ready state is designed to give a fast transition time to TX or RX state with reasonable current consumption. In this mode the crystal oscillator remains enabled reducing the time required to switch to TX or RX mode by eliminating the crystal start-up time. An automatic sequencer will put the chip into RX or TX from any state. It is not necessary to manually step through the states. Although it is not shown in the diagram, any of the lower power states can be returned to automatically after RX or TX.

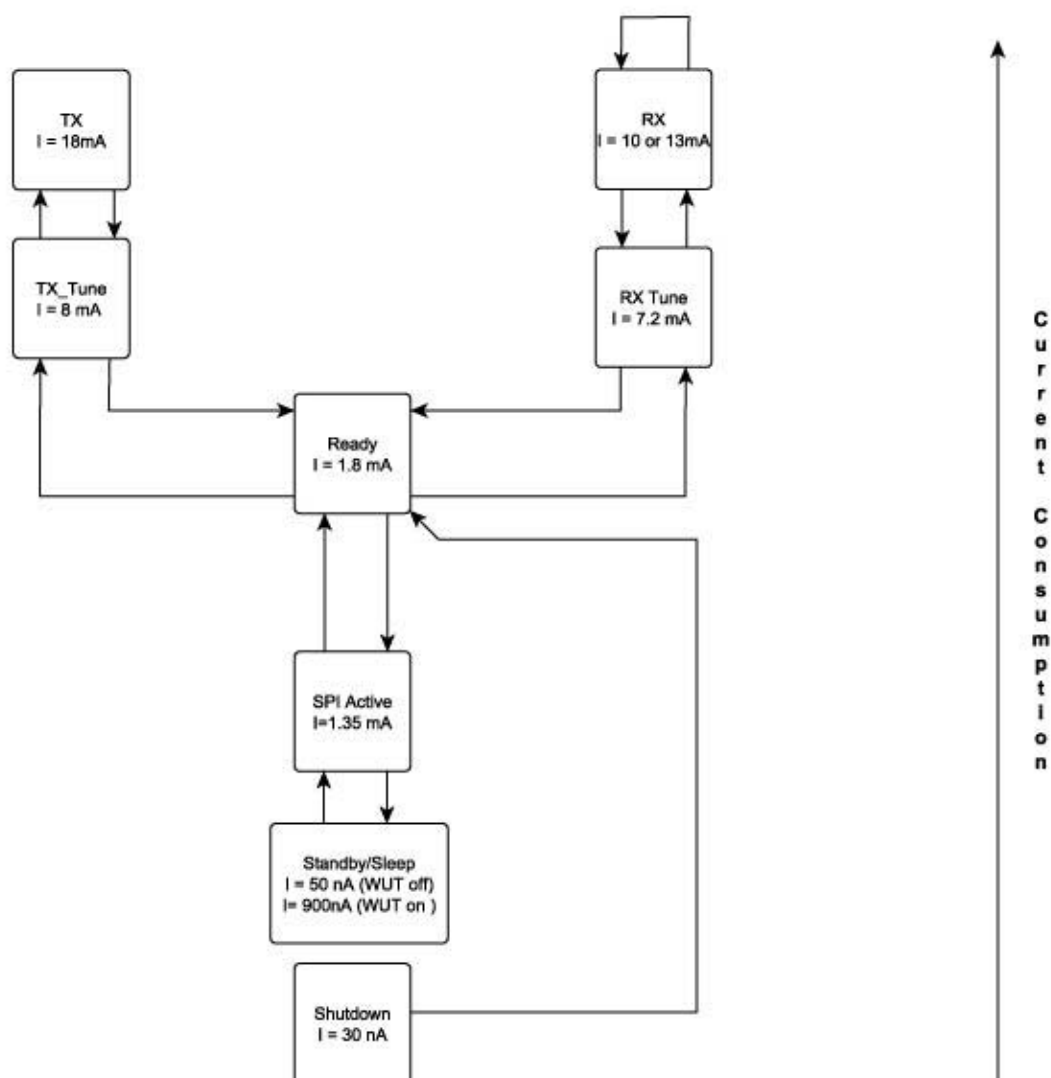


Figure 16. Operational States and Current Consumption

Table 5. Switching Times between Radio States

State/Mode	Response Time to	
	TX	RX
Shutdown	15 ms	15 ms
Sleep*	500 μ s	500 μ s
SPI Active	340 μ s	340 μ s
Ready	110 μ s	120 μ s
TX Tune	60 μ s	125 μ s
RX Tune	120 μ s	84 μ s
TX	130 μ s	132 μ s
RX	120 μ s	108 μ s

***Note:** While the chip is in sleep state, the NSEL pin has to stay in high state. If the host processor is not able to provide this during sleep, a pullup resistor can be necessary on the NSEL pin.

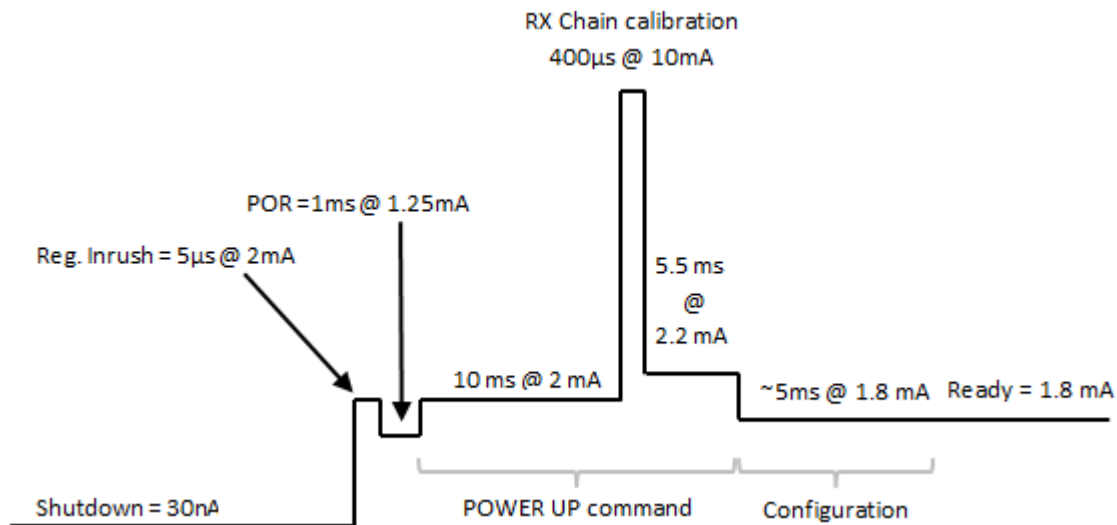


Figure 17. Supply Current versus Time Diagram from Shutdown to RF initialized Ready State

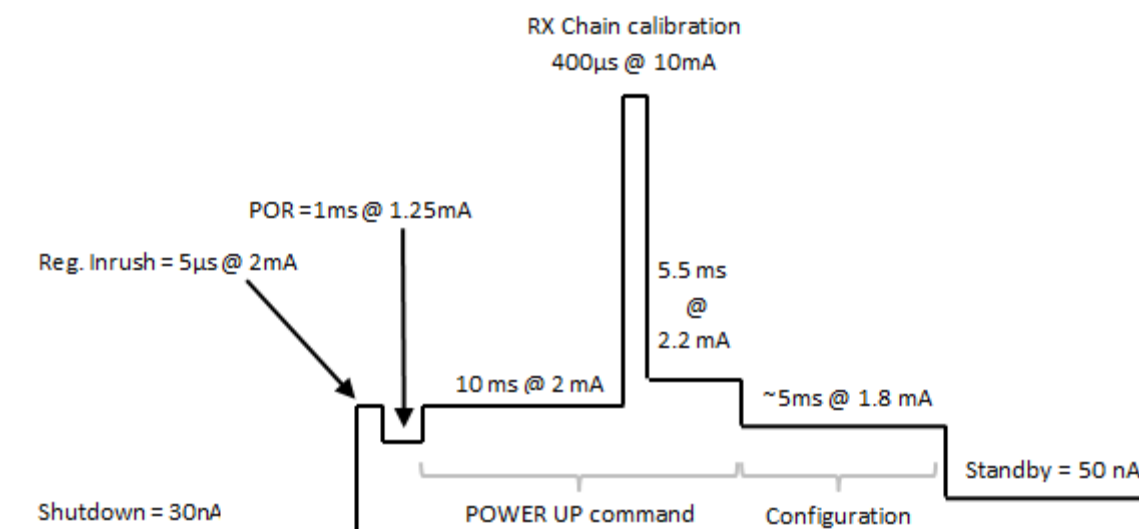


Figure 18. Supply Current versus Time Diagram from Shutdown to Standby State

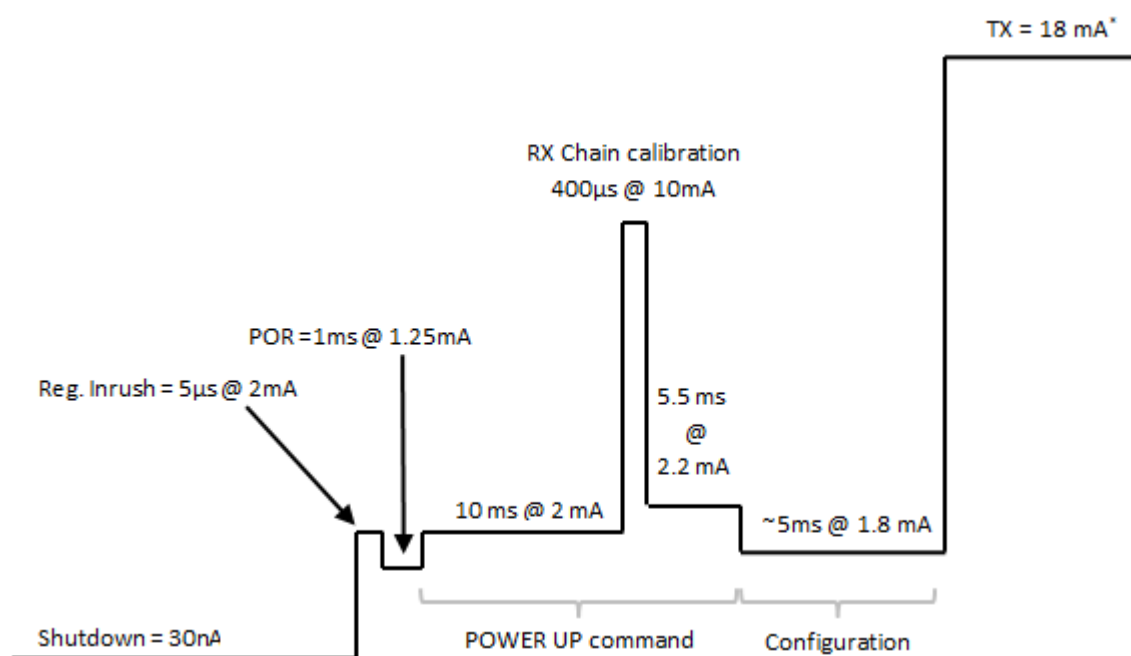


Figure 19. Supply Current versus Time Diagram from Shutdown to TX State

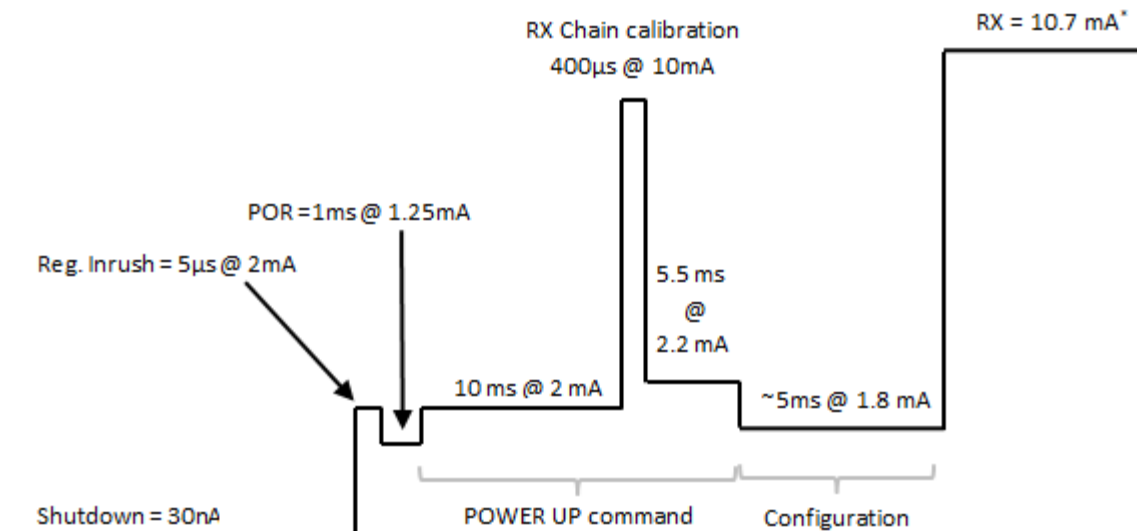


Figure 20. Supply Current versus Time Diagram from Shutdown to RX State

8.2. Radio Chip Waking Up

First, the radio is in the off state. After the SDN pin is pulled low, the radio wakes up and performs a Power on Reset which takes a maximum of 6 ms (900 μ s typical at room temperature) until the chip is ready to receive commands on the SPI bus. The GPIO1 pin goes high when the radio is ready for receiving SPI commands. During the reset period, the radio cannot accept any SPI commands. There are two ways to determine if the chip is ready to receive SPI commands after a reset event. Either use a timer in the host microcontroller to wait or 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 the GPIO1 to the high state.

Next, the radio device has to be sent to active mode by issuing a "POWER_UP" command via the SPI interface which takes approximately 15 ms to be completed. It can be monitored in three ways. If the command is completed either the GPIO1 pin of the radio goes low by issuing the command and the radio sets it to high state or the nIRQ pin is asserted or the host MCU can monitor CTS over the SPI.

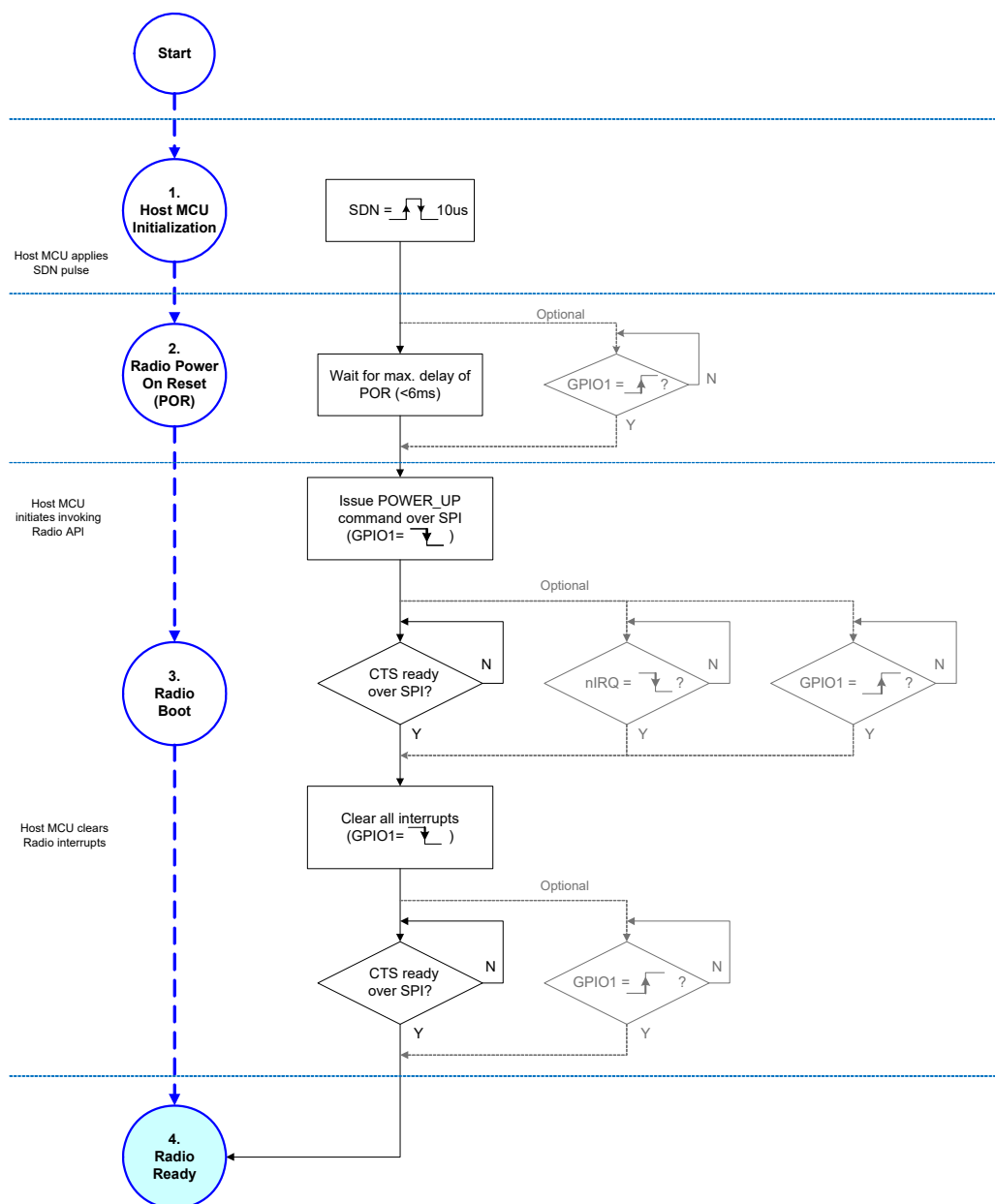


Figure 21. Radio Wake Up Process

8.3. Radio Initialization with Generated Radio Configuration File

8.3.1. Radio Initialization with RF Parameters

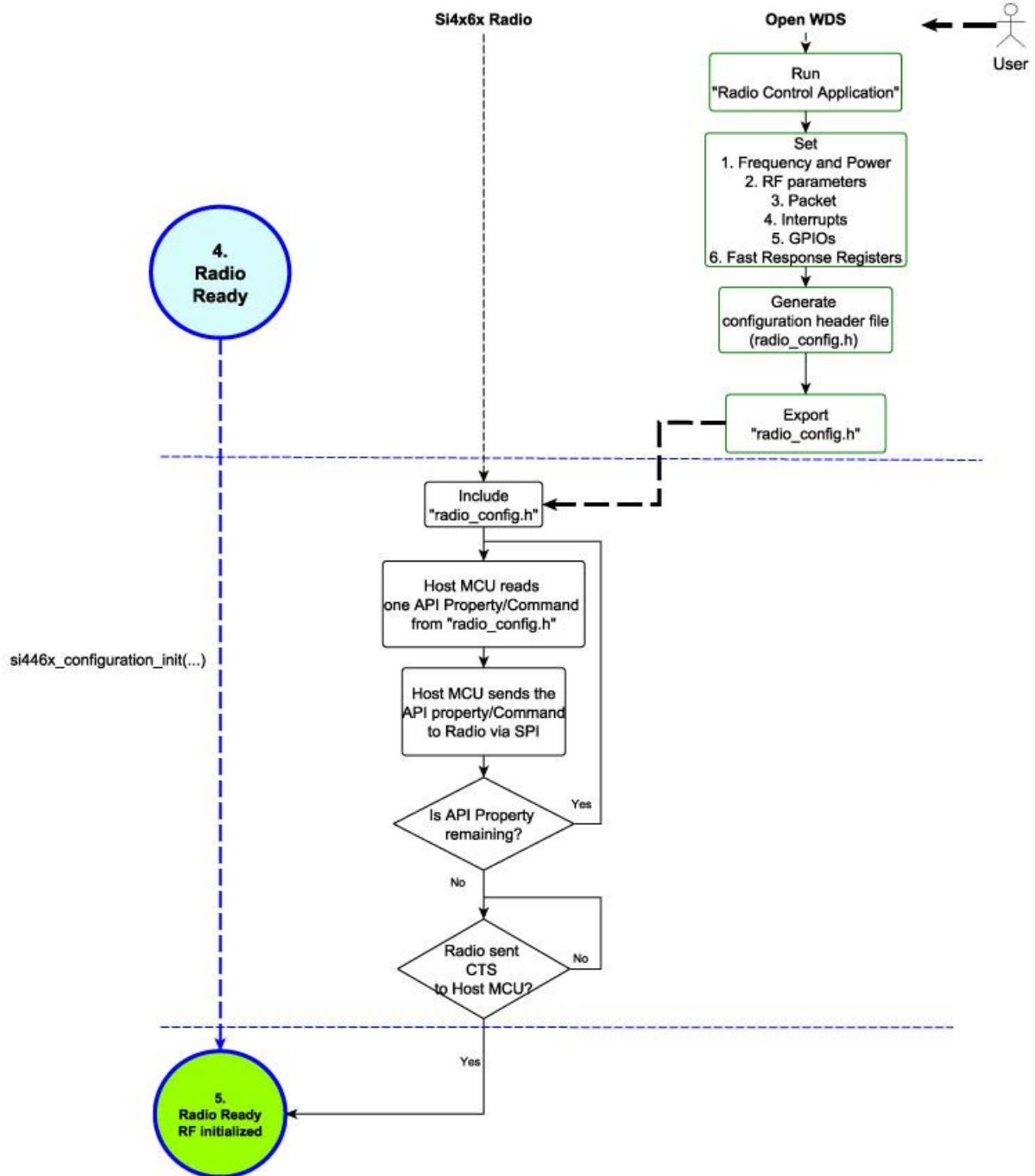


Figure 22. Radio RF Initialization Process

The radio parameter configuration process can be accomplished by using the Wireless Development Studio (WDS). After the required parameters are given to the radio configuration application, the WDS creates the configuration data based on these parameters. If the Launch IDE option is selected, the WDS generates a `radio_config.h` header file that contains the configuration data. This header file contains all the information needed by the application to configure the radio properly. This information includes the parameters of the RF link such as the modulation type, channel bandwidth, data rate, center frequency, crystal tolerance, crystal capacitor bank value, modulation source, CRC calculation and sync word setting. For more complete information on WDS and EZConfigPRO usage, refer to the WDS User's Guide.

8.3.2. Generated Radio Configuration File

The configuration file is automatically generated by the "Radio Control Application" tool. It is interpreted as a C-header file called "radio_config.h" and it has four sections. The first two sections are intended for the users and allow them to see the exact values of the API properties. The last two sections are specifically intended for the example projects. The structure of the header file is shown in Figure 23.

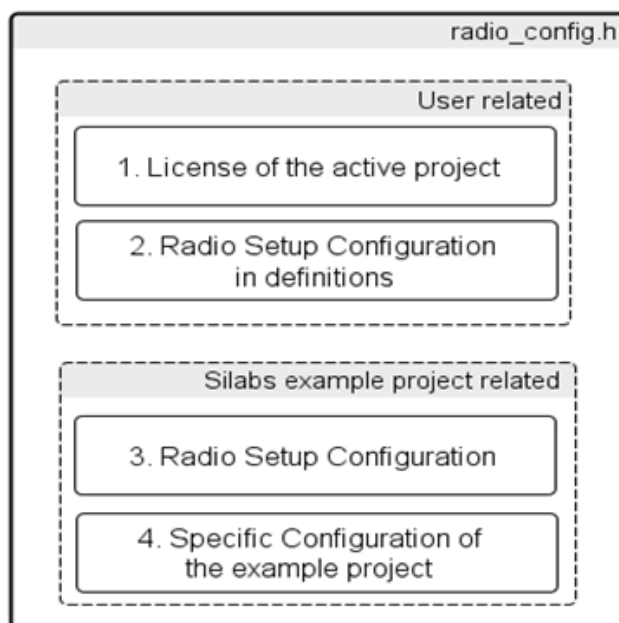


Figure 23. Radio_config.h File Structure

The "License of the Active Project" section consists of numerous commented lines about the license.

```
/**
```

```
* Copyright 2012 Silicon Laboratories, Inc.
```

```
* http://www.silabs.com
```

```
*
```

```
* @file radio_config.h
```

```
*
```

```
* H File Description:
```

```
* @brief Radio configuration macros and definitions.
```

```
*
```

```
* Project Name: EZRPRO_Examples
```

```
*
```

```
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
```

AN633

* PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
* HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
* CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
* SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

*

* This software must be used in accordance with the End User License Agreement.

The "Radio Setup Configuration in Definitions" section is the list of initialization commands that are sent to the radio over the SPI interface. The structure of one element of the list is shown below. The comment lines describe how the C definition configures the dedicated API properties. The C define line stands for the initialization command.

```
/* Set Multiple Properties starting with MODEM_MOD_TYPE
```

```
* Starting Location: 0x2000
```

```
* Number of properties: 3
```

```
* Defaults: 0x02 0x80 0x07
```

```
* Descriptions:
```

```
* MODEM_MOD_TYPE - Modulation Type.
```

```
* Bit[7] : TX_DIRECT_MODE_TYPE : 0
```

```
* Bit[6:5] : TX_DIRECT_MODE_GPIO [1:0] : 0
```

```
* Bit[4:3] : MOD_SOURCE[1:0] : 0
```

```
* Bit[2:0] : MOD_TYPE[2:0]: 3
```

```
* MODEM_MAP_CONTROL - Controls bit mapping.
```

```
* Bit[7] : enmanch : 0
```

```
* Bit[6] : eninv_rxbit : 0
```

```
* Bit[5] : envinv_txbit : 0
```

```
* Bit[4] : envinv_fd : 0
```

```
* Bit[3] : envinv_adcq : 0
```

```
* MODEM_DSM_CTRL - DSM control.
```

```
* Bit[7] : dsmclk_sel : 0
```

```
* Bit[6] : dsm_mode : 0
```

```
* Bit[5] : dsmdt_en : 0
```

```
* Bit[4] : dsmdttp : 0
```

```
* Bit[3] : dsm_rst : 0
```

```
* Bit[2] : dsm_lsb : 1
```

```
* Bit[1:0] : dsm_order :3
```

```
*/
```

```
#define RADIO_CONFIGURATION_SET_PROPERTIES_MODEM_MOD_TYPE 0x11, 0x20, 0x03, 0x00, 0x03,  
0x00, 0x07
```

The commented lines explain which API property/properties will be overwritten with new value(s). For example, this definition is responsible for initializing three API properties of the radio at one time, "MODEM_MOD_TYPE", "MODEM_MAP_CONTROL", and "MODEM_DSM_CTRL". The format of the definition is as follows:

- The first byte is the command ID of the "SET_PROPERTY" API command.
- The next three bytes are the requirements of the command:
 - MSB of the starting location of the API property
 - Number of the immediate adjacent API properties
 - LSB of the starting location of the API property
- Finally, the values of the properties set by the command.

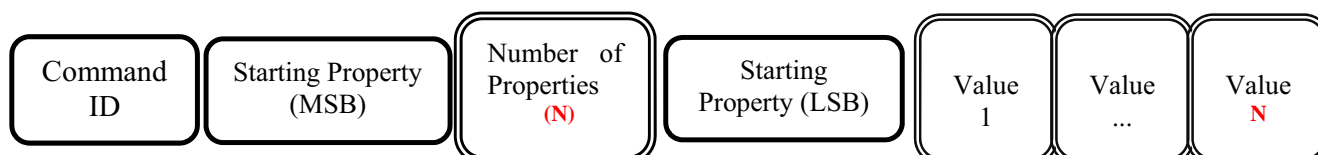


Figure 24. Structure of "SET_PROPERTY" API Command

The "Radio Setup Configuration" section is intended only for the example project. This part only has a C-like structure, called "Radio_Setup_Configuration_Array", that contains the previously mentioned definitions with the modification that the first element is the length of the API command. The format of the definition remains almost the same but the length field is added. The importance of the extended structure is to build the appropriate format of the input parameter for the 'Si446x_configuration_init(...)' function. The format of a line from the array is as follows:

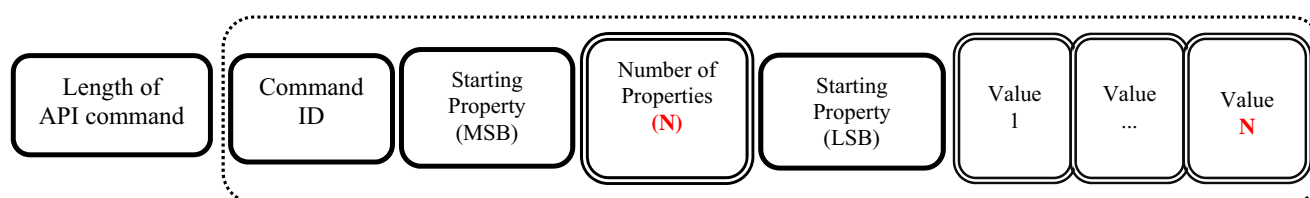


Figure 25. One Element of the Radio Setup Configuration Section

The "Specific Configuration of the Example Project" section is intended for the example project. It contains some specific definitions for the example application such as which channel is selected either for the transmission or for the reception. The content of the custom packet is initialized in this section. It also contains WDS calculator-related information in the commented lines.

```
/**
 * crys_freq  xtaltol  ifmode  hipfm_Ch_Fil  OSRtune  chfil_sw_AFC  ant_div  pm_pattern
 * 30000000  20      2      1          0      0      0      0
 *
 * mod_type   Rsymb(sps)  Fdev(Hz)  OOK_BW(Hz)  Manchester  afc_en  Rb_error  Chip_Version
 * 1          40000     83333.3  350000     0          0      0      2
 *
 * fc(MHz)    API_TC     fhst     hi_pfm_cp   BER_mode   raw_demod  dsources  hi_pfm_div
 * 915        27      0      0          0          0      0      1
 */
#define CUSTOM_PACKET_TX_CONTENT {SiliconLabs}
#define CUSTOM_PACKET_RX_CONTENT {SiliconLabs}
#define CUSTOM_CHANNEL_NUMBER_TX 0
#define CUSTOM_CHANNEL_NUMBER_RX 0
```

9. Example Projects and Software Layers

9.1. Software Layers

In all of the sample projects, the layered software approach is followed. There is a distinct scope for each software module, and all modules can communicate through each other's API functions. The software modules are separated and focused to cover one specific task. Figure 26 shows the software layers and its relations.

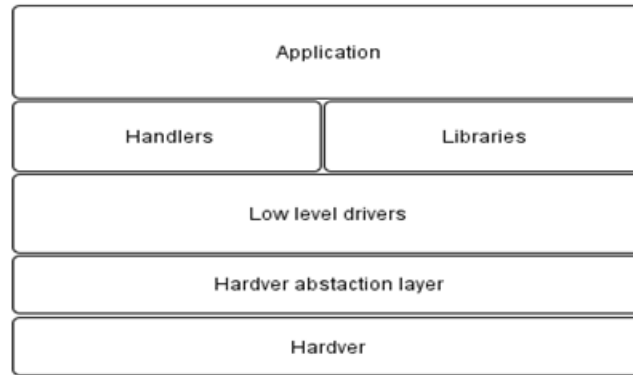


Figure 26. Software Layers of the Example Codes

9.2. Radio Initialization in the Software Layers Perspective

Using the software layer approach, the example project can be understood easily. Each and every layer has its own responsibility. If the upper layer, e.g. the "Application", wants to configure the hardware platform including the host microcontroller and also the radio chip, it simply calls the hardware initial routine. The radio chip initialization is started with a power on reset. The radio module sends a request to the si446x radio driver to reset the chip. Thereafter, the driver forwards the request to the hardware abstraction layer that pulls down the SDN pin to perform the power on reset. After the POR, the host MCU needs to send all the API properties to the radio via SPI interface that means the "radio setup configuration" of the radio_config.h header file needs to be processed line by line. The whole process of sending one API property and checking whether the radio is ready to receive the next property is a repetitive task is represented by a configuration loop. Finally, host MCU clears all the pending interrupts of the radio that is initialized to ready in RF perspective.

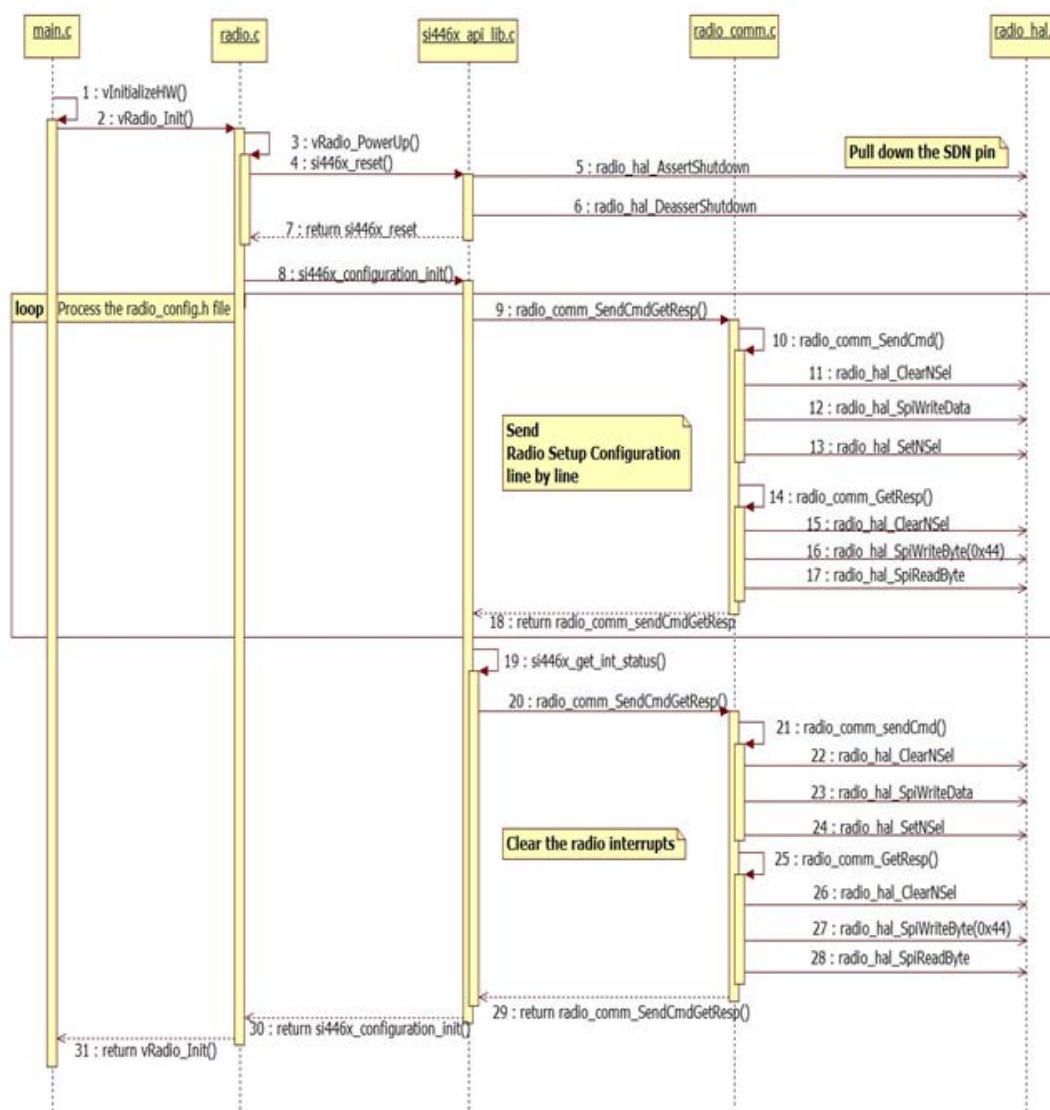


Figure 27. Function Calls During the Radio Initialization

9.3. Directory Structure of the Example Project

All sample source code has a common directory structure with separated source and project files to ease the understanding of the individual modules. For every sample project, the following directories and files can be found in the main directory:

- **bin**—Contains the SiLabs project files for Keil and SDCC compilers and the Makefile if the make tool is used instead.
- **doc**—Doxygen-generated documentation based on comments inside the source files in html format.
- **out**—The outputs of the compilation process are sent to this folder. After successful compilation, this directory contains files such as the hex file, the linker output, and the OMF file.
- **src**—Directories containing the source files.
 1. application
 2. drivers
- **Doxyfile**—This file contains the Doxygen documentation generator settings.

- **Cleanup.bat**—Batch file used to delete all files generated during build process.

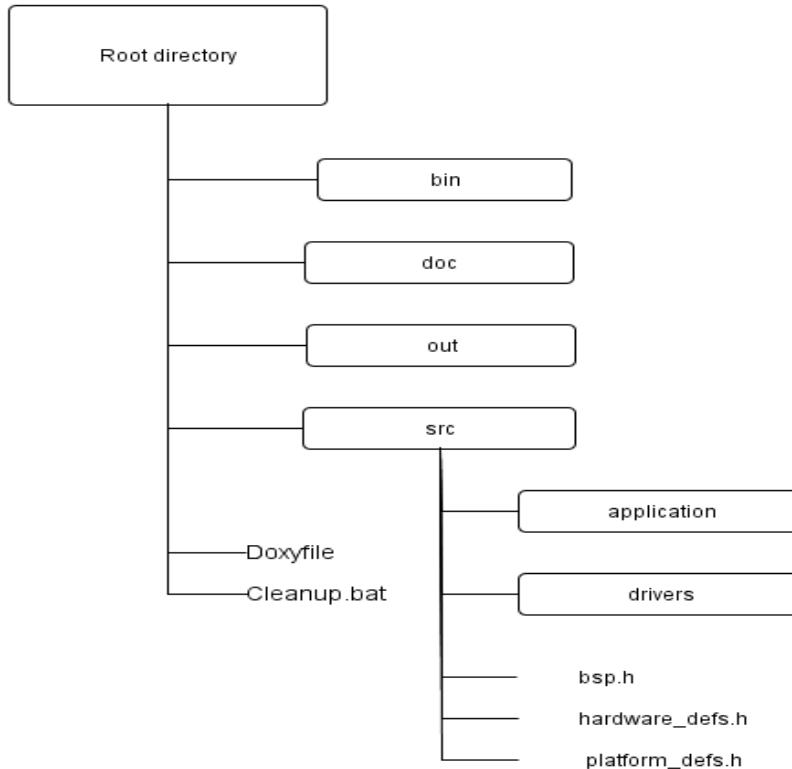


Figure 28. Directory Structure of the Example Project

The individual software modules are separated into several source files. The sample projects contain one header file (`bsp.h`) that is included in the source files and collects the individual headers that need to be included. Under “src” folder, the “application” folder contains application-related sources. Common modules (e.g., handlers, drivers) are located under the “driver” directory.

9.4. Common Software Modules

In the modules hierarchy, the common software modules (CSM) are located between the application and the hardware layers. The CSM is a set of interfaces that provide possible options for controlling various peripherals on modular HW platforms. Registers can be initialized with pre-configured settings and peripherals can be enabled to start/stop their own processing. The major tasks of these software modules are to initialize the hardware elements and control its behaviors. The principle of their installation is to provide a façade for the upper layers. Functionally, the User Application at the top of the hierarchy can be independent of the hardware and its logical operation can remain unchanged even if the hardware has been modified later. It can be adapted to any device without encountering difficulties. All the modules in the following subsections except the human-machine interface module are primarily responsible for handling the dedicated internal peripherals such as the IO, timers, SPI, and PCA. The HMI holds the peripherals together so it gives a higher abstraction level to the User Application in the form of handlers.

9.4.1. Common Software Modules Location

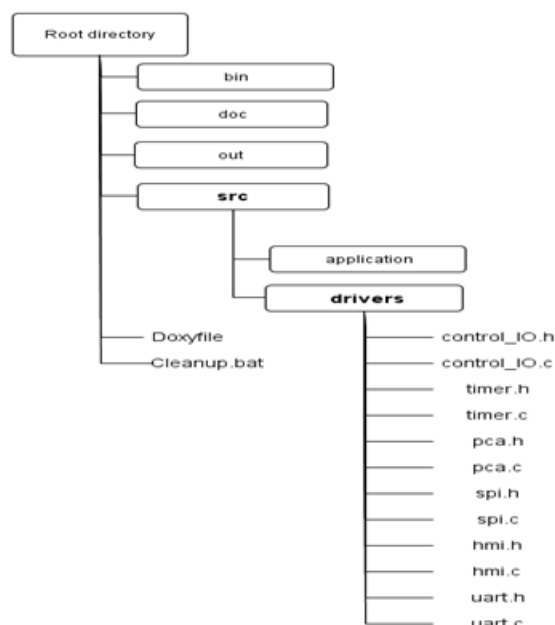


Figure 29. Location of the Common Software Modules

9.4.2. Input/Output Control Module

The Input/Output (I/O) control-related source files, called `control_IO.h` and `control_IO.c`, are located in the `/src/drivers/` folder. The module handles the port initializations for the physical HW platform such as LEDs, push-buttons, or buzzer. For example, the module can set the state of the LEDs and read the status of the selected push-buttons.

Function Name: void `vCio_InitIO`(void)
Description: This function is used to initialize specific IO port for LED and PB.
Return Value: None
Note: It has to be called from the initialization section.

Function Name: void `vCio_SetLed`(uint8_t biLedNum)
Description: This function is used to switch the selected LED on.
Input Parameter(s): biLedNum : Number of the LED to be switched on (1-4).
Return Value: None

Function Name: void `vCio_ClearLed`(uint8_t biLedNum)
Description: This function is used to switch the selected LED off.
Input Parameter(s): biLedNum : Number of the LED to be switched off (1-4).
Return Value: None

Function Name: BIT `gCio_GetPB`(uint8_t biPbNum)
Description: This function is used to read the status of the selected push-button.
Input Parameter(s): biPbNum : Number of the push-button to be switched on (1-4).
Return Value: State of the selected PB.

9.4.3. Timer Peripheral Module

The timer-related source files, called timer.h and timer.c, are located in the /src/drivers/ folder. This module handles two 16-bit timers, timer2 and timer3. The most accurate timing interval can be calculated from the frequency of the system clock, which is generally 24.5 MHz. External clock sources can be selected as timer input and the required timing frequency can be adjusted thoroughly with several different prescalers. In general, the timer files are set to a frequency of 1 kHz (1 ms). By using the timer with 1 ms settings, timeouts that are a multiple of 1 ms can be easily implemented. Timer-related operations provide options to start or stop counting. Additionally, interrupts can be generated when the low byte of the timer overflows. Timers can also be checked for expiration.

Function Name: void **vTmr_StartTmr2**(uint8_t biPrescaler, uint16_t wiPeriod, uint8_t biltEnable, uint8_t biExtClkSel)
Description: This function is used to start Timer 2 in the specified mode.
biPrescaler : Prescaler value of timer .
(use predefined constants: bTmr_Tmr2One_c, bTmr_Tmr2Both_c)
wiPeriod : The duration of the timing.
Input Parameter(s): biltEnable : Enables timer IT if TRUE, disables it if FALSE.
biExtClkSel External clock select.
(use predefined constants: bTmr_TxCLK_00_c etc.)
Return Value: None.

Function Name: BIT **gTmr_Tmr2Expired**(void)
Description: This function is used to check if Timer 2 is expired.
Return Value: True if timer is expired (also stops the timer).
Note: Function clears the IT status flag as well.

Function Name: void **vTmr_StartTmr3**(uint8_t biPrescaler, uint16_t wiPeriod, uint8_t biltEnable, uint8_t biExtClkSel)
Description: This function is used to start Timer 3 in the specified mode.
biPrescaler : Prescaler value of timer.
(use predefined constants: bTmr_Tmr3One_c, bTmr_Tmr3Both_c)
wiPeriod : The duration of the timing.
Input Parameter(s): biltEnable : Enables timer IT if TRUE, disables it if FALSE.
biExtClkSel : External clock select.
(use predefined constants: bTmr_TxCLK_00_c etc.)
Return Value: None

Function Name: BIT **gTmr_Tmr3Expired**(void)
Description: This function is used to check if Timer 3 is expired.
Return Value: True if timer is expired (also stops the timer).
Note: Function clears the IT status flag as well.

9.4.4. Programmable Counter Array Module

The programmable counter array (PCA)-related source files, called `pca.h` and `pca.c`, are located in the `/src/driver/` folder. This module initializes the PCA, which creates beeping sounds from the buzzer. The time-base source of the PCA can be selected. Interrupts can be generated when the lower byte of the counter overflows. PWM-mod cycle length also can be selected to modify the frequency of the tweeting sound.

Function Name: void `vPca_InitPcaTmr`(uint8_t biPulseSelect, uint8_t biPcaTmrItEnable, uint8_t biCycleLengthSelect)

Description: This function is used to start Timer 2 in the specified mode.

Input Parameter(s): biPulseSelect : Selects time-base source of PCA .
(use predefined constants: `bPca_PcaCps_000_c` etc.)
biPcaTmrItEnable : Enables PCA timer IT if TRUE, disables it if FALSE.
biCycleLengthSelect : PWM-mode cycle length select
(use predefined constants: `bPca_PwmClSel_00_c`, etc.).

Return Value : None

9.4.5. Serial Peripheral Interface Module

The serial peripheral interface (SPI)-related source files, called `spi.h` and `spi.c`, are located in the `/src/driver/` folder. This module is the most essential because it enables a connection to the radio via the SPI bus. The radio can be controlled by its built-in application programming interface. Communication with the radio is based on sending commands to the API and receiving responses from the API. To enable the SPI interface, the SPI port must be enabled and associated to the crossbar. The directions of the SCK, MISO, and MOSI ports have to be configured properly on the IO port. Finally, the default states of the pins have to be set correctly. Since several devices can be connected to the same SPI bus, the NSEL pin of the selected device is activated during communication. Because the commands to be sent to the API are sequences of bytes, the module has to be able to send and receive continuous byte stream. There are some cases when either reading a single byte directly from the MISO or writing specified number of bits directly to the MOSI is necessary. In order to cover these kinds of cases, bitbang read/write methods have been also implemented.

Function Name: uint8_t `bSpi_ReadWriteSpi0`(uint8_t biDataIn)

Description: This function is used to read/write one byte from/to SPI0.

Input Parameter(s): biDataIn : Data to be sent.

Return Value: Read value of the SPI port after writing on it.

Function Name: uint8_t `bSpi_ReadWriteSpi1`(uint8_t biDataIn)

Description: This function is used to read/write one byte from/to SPI1.

Input Parameter(s): biDataIn : Data to be sent.

Return Value: Read value of the SPI port after writing on it.

Function Name: void `vSpi_WriteDataSpi0`(uint8_t biDataInLength, uint8_t *pabiDataIn)

Description: This function is used to send data over SPI0 no response expected.

Input Parameter(s): biDataInLength : The length of the data.
*pabiDataIn : Pointer to the first element of the data.

Return Value: None

AN633

Function Name: void **vSpi_WriteDataSpi1**(uint8_t biDataInLength, uint8_t *pabiDataIn)
Description: This function is used to send data over SPI1 no response expected.
Input Parameter(s): biDataInLength : The length of the data.
*pabiDataIn : Pointer to the first element of the data.
Return Value: None

Function Name: void **vSpi_ReadDataSpi0**(uint8_t biDataOutLength, uint8_t *paboDataOut)
Description: This function is used to read data from SPI0.
Input Parameter(s): biDataOutLength :The length of the data.
Output Parameters(s): *paboDataOut : Pointer to the first element of the response.
Return Value: None

Function Name: void **vSpi_ReadDataSpi1**(uint8_t biDataOutLength, uint8_t *paboDataOut)
Description: This function is used to read data from SPI1.
Input Parameter(s): biDataOutLength : The length of the data.
Output Parameters(s): *paboDataOut : Pointer to the first element of the response.
Return Value: None

Function Name: void **vSpi_EnableSpi0**(void)
Description: This function is used to enable SPI0 and associate to XBAR.
Return Value: None

Function Name: void **vSpi_EnableSpi1**(void)
Description: This function is used to enable SPI1 and associate to XBAR.
Return Value: None

Function Name: void **vSpi_DisableSpi0**(void)
Description: This function is used to disable SPI0 and disconnect from XBAR.
Return Value: None

Function Name: void **vSpi_DisableSpi1**(void)
Description: This function is used to disable SPI1 and disconnect from XBAR.
Return Value: None

Function Name: void **vSpi_ClearNselSpi0**(uint8_t biSelectDevice)
Description: This function is used to pull down nSEL of the selected device on SPI0.
Input Parameter(s): biSelectDevice Selected device
Return Value: None

Function Name: void **vSpi_ClearNselSpi1**(uint8_t biSelectDevice)
Description: This function is used to pull down nSEL of the selected device on SPI1.
Input Parameter(s): biSelectDevice Selected device
0 - DUT
2 - EEPROM
3 - MCU2
Return Value: None

Function Name: void **vSpi_SetNselSpi0**(uint8_t biSelectDevice)
Description: This function is used to pull up nSEL of the selected device on SPI0.
Input Parameter(s): biSelectDevice : Selected device
Return Value: None

Function Name: void **vSpi_SetNselSpi1**(uint8_t biSelectDevice)
Description: This function is used to pull up nSEL of the selected device on SPI1.
Input Parameter(s): biSelectDevice: Selected device
0 - DUT
2 - EEPROM
3 - MCU2
Return Value: None

Function Name: uint8_t **bSpi_ReadByteBitbangSpi0**(void)
Description: This function is used to read one byte from SPI0 using bitbang method.
Return Value: Read byte

Function Name: uint8_t **bSpi_ReadByteBitbangSpi1**(void)
Description: This function is used to read one byte from SPI1 using bitbang method.
Return Value: Read byte

Function Name: void **vSpi_WriteBitsBitbangSpi0**(uint8_t biDataIn, uint8_t biNumOfBits)
Description: This function is used to write specified number of bits to SPI0 using bitbang method.
Input Parameter(s): biDataIn : Input byte of data bits
Output Parameters(s): biNumOfBits : Number of bits to be written to SPI
Return Value: None

Function Name: void **vSpi_WriteBitsBitbangSpi1**(uint8_t biDataIn, uint8_t biNumOfBits)
Description: This function is used to write specified number of bits to SPI1 using bitbang method.
Input Parameter(s): biDataIn : Input byte of data bits
Output Parameters(s): biNumOfBits : Number of bits to be written to SPI.
Return Value: None

9.4.6. Human Machine Interface Module

The human machine interface (HMI)-related source files, called hmi.h and hmi.c, are located in the /src/driver/ folder. In order to use this module, the required handlers need to be initialized at the very beginning of the program. The status of the various hardware components must be checked in order to have a common cyclic mechanism. In addition, it is vital that a 1 ms interrupt-based cycle run in the background to serve the different handlers. Using the LED handler, states of LEDs can be set and cleared either separately or together.

Using the button handler, statuses of the push-buttons can be read. If multiple button events happen simultaneously, they can be stored to be handled later. The last pushed button event is always available first amongst the un-handled events. Using the buzzer related sub-interface, the state of buzzer can be changed to the required one.

Function Name: void **vHmi_InitLedHandler**(void)
Description: This function is used to initialize the Led handler.
Return value: None
Note: Must be called from the initialization section.
Re-initialization of LED Handler supported by the extended HMI driver.

Function Name: void **vHmi_ChangeLedState**(eHmi_Leds qiLed, eHmi_LedStates qiLedState)
Description: This function is used to change state of selected Led.
Input Parameter(s): qiLed : Led to change its state
qiLedState : New state of qiLed
Return Value: None

Function Name: void **vHmi_ChangeAllLedState**(eHmi_LedStates qiLedState)
Description: This function is used to change state of all Leds.
Input Parameter(s): qiLedState : New state of all the Leds
Return Value: None

Function Name: void **vHmi_ClearAllLeds**(void)
Description: This function is used to force all Leds to off immediately.
Return Value: None

Function Name: void **vHmi_LedHandler**(void)
Description: This function is used to handle Led management.
Return Value: None

Function Name: void **vHmi_InitPbHandler**(void)
Description: This function is used to initialize push-button handler.
Return Value: None
Note: It has to be called from the initialization section.
Re-initialization of LED Handler supported by the extended HMI driver.

Function Name:	BIT gHmi_PbIsPushed (uint8_t *boPbPushTrack, uint16_t *woPbPushTime)
Description:	This function is used to check if any of the push-buttons are pushed.
Output Parameters(s):	*boPbPushTrack : Read value of pushed button. *woPbPushTime : Push time of pushed button.
Return Value:	Pushed state of push-buttons
Function Name:	bool gHmi_IsPbUnHandled (void)
Description:	This function is used to check if there is unhandled push-buttons event.
Return Value:	True if there is unhandled push-button event.
Function Name:	uint8_t bHmi_PbGetLastButton (uint16_t *woPbPushTime)
Description:	This function is used to read last pushed button(s), push track holder is erased if button(s) was already released.
Output parameters(s):	*woPbPushTime : Push time of pushed button.
Return Value:	Push track holder of last pushed button(s)
Function Name:	void vHmi_PbHandler (void)
Description:	This function is used to handle push-button management.
Return Value:	None
Function Name:	void vHmi_ShowPbOnLeds (void)
Description:	This function is used to show the actual state of the push-buttons on the Leds.
Return Value:	None
Function Name:	bool gHmi_SwStateHandler (void)
Description:	This function is used to handle switch state change.
Return Value:	True if state of switches has changed
Function Name:	uint8_t bHmi_GetSwState (void)
Description:	This function is used to handle give the state
Return Value:	State
Function Name:	void vHmi_InitBuzzer (void)
Description:	This function is used to initialize the buzzer operation.
Return Value:	None
Note:	It has to be called from the initialization section.
Function Name:	void vHmi_ChangeBuzzState (eHmi_BuzzStates qiBuzzState)
Description:	This function is used to change the state the buzzer.
Input Parameter(s):	qiBuzzState : New state of the buzzer
Return Value:	None

Function Name: void **vHmi_BuzzHandler**(void)
Description: This function is used to handle buzzer management.

9.4.7. UART interface Module

The UART related source files, called `uart.h` and `uart.c`, are located in the `/src/driver/` folder. In order to use this module, the functionality needs to be initialized at the very beginning of the program.

Bytes can be sent and received as well and the functionality uses the `uart` interrupt service routine.

Function Name: uint8_t **Comm_IF_RecvUART**(uint8_t * byte)
Description: This function is used to receive bytes from UART.
Output Parameters(s): *byte Pointer to the first element of the incoming data.
Return Value: True if there is an incoming data otherwise FALSE.

Function Name: uint8_t **Comm_IF_SendUART**(uint8_t byte)
Description: This function is used to send bytes through UART.
Input Parameters(s): byte : data to be sent
Return Value: True if sending data completed successfully otherwise FALSE.

Function Name: void **Comm_IF_EnableUART**(void)
Description: Enable and set the UART0 peripheral
Return Value: None

9.4.8. Size Optimization of the Common Software Modules

To optimize the code size of the common software modules and consequently the example source codes, software switches must be introduced in almost every module. By enabling these switches, new functions can be added to the whole project to compile. If the switch is not defined at the beginning of the bsp.h header file, then only the basic features can be used, which is barely sufficient for most of the example codes.

The rest of the module's features can be added to the project by defining the following switches:

- `TIMER_DRIVER_EXTENDED_SUPPORT`
- `SPI_DRIVER_EXTENDED_SUPPORT`
- `HMI_DRIVER_EXTENDED_SUPPORT`
- `UART_DRIVER_EXTENDED_SUPPORT`

The control_IO and PCA modules are quite simple, so there is no need to use driver extension in these cases.

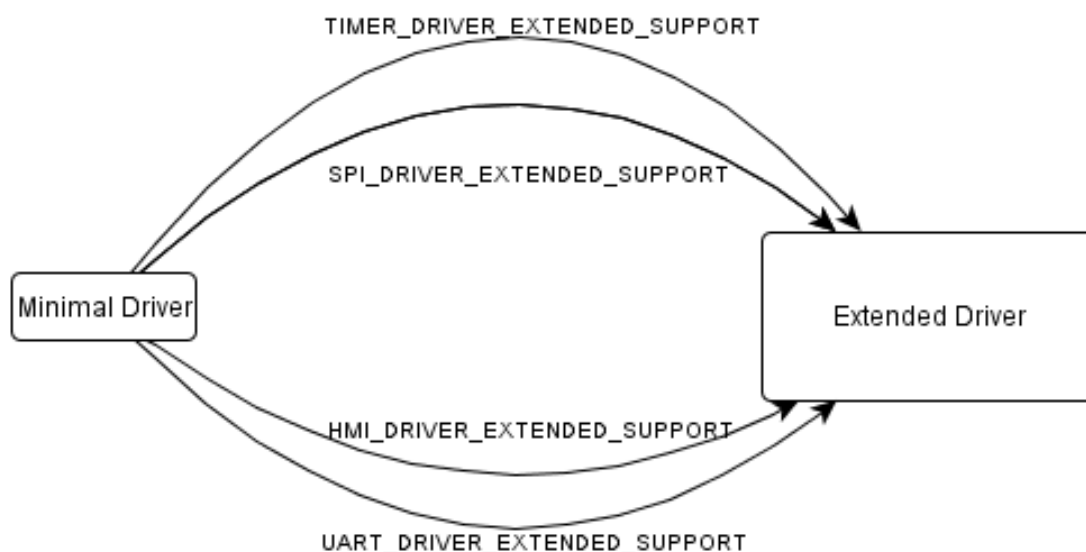


Figure 30. Usage of Software Switches

Table 6 shows which module can support the driver expansion feature:

Table 6. Size Optimization Possibilities for Common Software Modules

Common Software Module	Software Switch	
	Minimal Driver	Extended Driver
Control IO	default	Not supported
Timer	default	<code>TIMER_DRIVER_EXTENDED_SUPPORT</code>
PCA	default	Not supported
SPI	default	<code>SPI_DRIVER_EXTENDED_SUPPORT</code>
HMI	default	<code>HMI_DRIVER_EXTENDED_SUPPORT</code>
UART	default	<code>UART_DRIVER_EXTENDED_SUPPORT</code>

Table 7 shows the comparison between the modules' sizes:

Table 7. Size Comparison of Common Software Modules

Common Software Module	Module Code Size [Byte]		Size Optimization [%]
	Minimal Driver	Extended Driver	
Control I/O	60	Not supported	—
Timer	60	146	58
PCA	29	Not supported	—
SPI	127	252	49
HMI	698	1295	46
UART	49	246	80

9.5. Radio Driver

The radio driver module resides in a low-level driver software layer. It is intended to provide a user-friendly and easy-to-use API to the radio functionality. It contains API functions and macro definitions for all radio commands. Constants can be found in the "EZRadioPRO API Documentation" zipped html file at <http://www.silabs.com/products/wireless/EZRadioPRO/Pages/Si446x.aspx>. Including this driver module into the software project makes the control of the radio chip easier than ever before through its comprehensive public API functions. The driver handles all the SPI communication with the chip, including the check for the CTS signal, and automatically reads the response from the chip. Thanks to the layered approach, it can be easily ported to other architectures and platforms, as it depends only on the Hardware Application Layer. This means it is enough to port the HAL for a given architecture in order to get the radio driver work. As introduced in the other drivers, the radio driver also can be compiled with different support types (Minimal, Extended or Full). Depending on which support type defined, the radio driver provides different level of API coverage. This way it provides a convenient way of managing the compile firmware size depending on the API functions usage and excludes the unused and, therefore, unnecessary functions.

9.5.1. Radio Driver Location

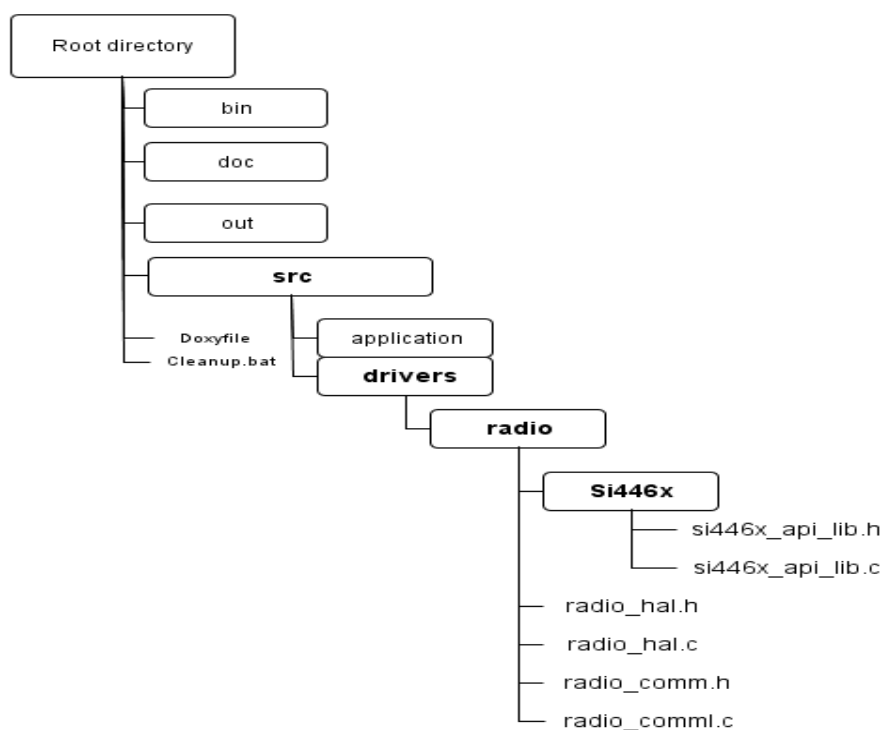


Figure 31. Location of the Radio Driver

9.5.2. Size Optimization of the Radio Driver

In order to optimize the code size of the common software modules and consequently the example project as well, software switches must be introduced in the radio driver. By activating the switches, new functions can be added to the radio driver. There are three ways of using it. If any of the radio driver switches is not defined at the beginning of the bsp.h header file, then only the basic features are used. It is sufficient for all example projects to work. The rest of the features can be added to the driver in two levels with the following switches:

- RADIO_DRIVER_EXTENDED_SUPPORT
- RADIO_DRIVER_FULL_SUPPORT

Table 8. Size Optimization Possibilities for Radio Driver

	Software Switch		
Driver	Minimal Driver	Extended Driver	Full Driver
Radio	default	RADIO_DRIVER_EXTENDED_SUPPORT	RADIO_DRIVER_FULL_SUPPORT

Table 9. Size Comparison of Radio Driver

	Module size [byte]		
Driver	Minimal Driver	Extended Driver	Full Driver
Radio	922	1094	1490

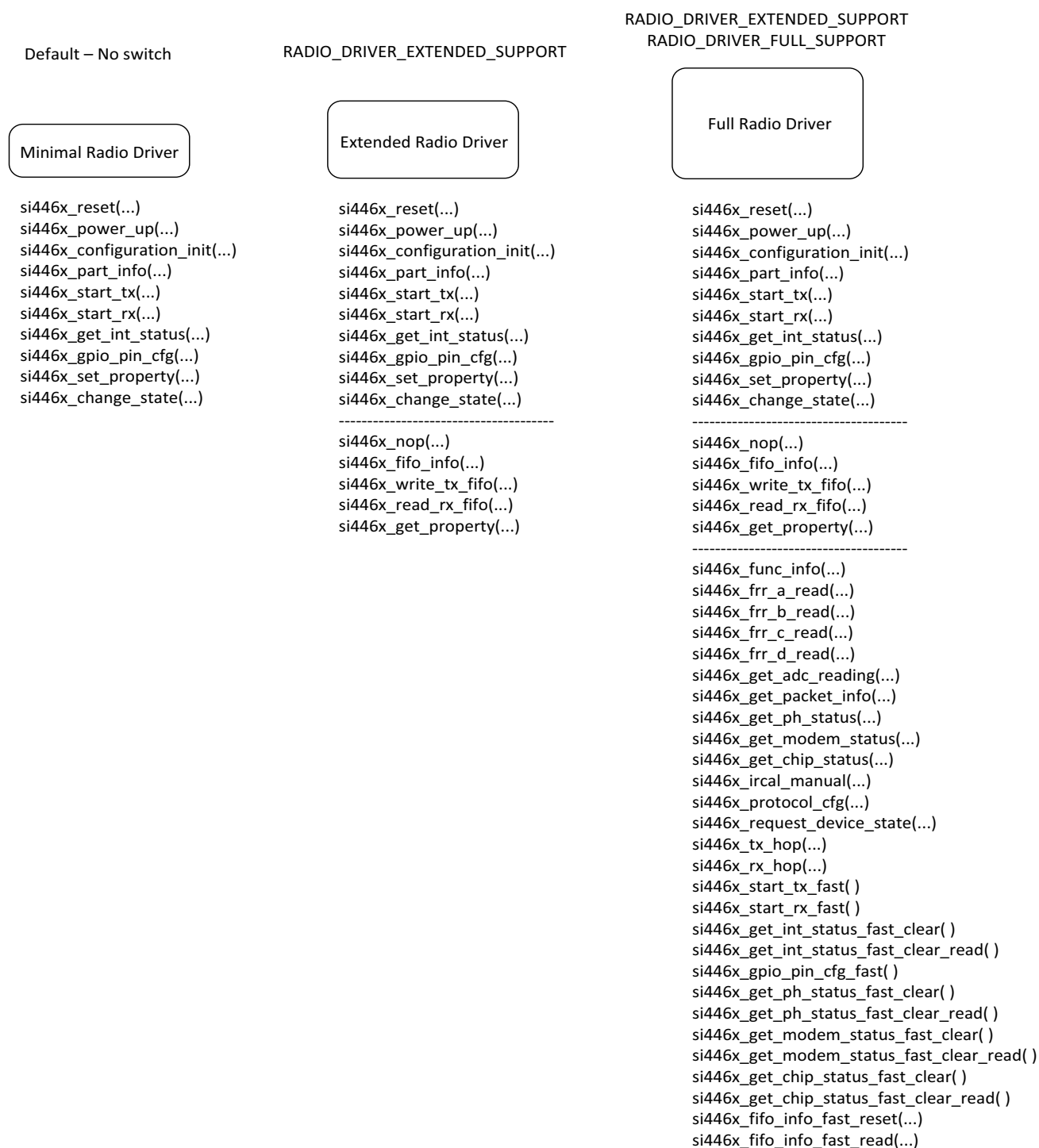


Figure 32. Usage of Radio Driver Switches

9.5.3. Minimal Radio Driver

- Function Name:** void **si446x_reset**(void);
- Description:** This functions is used to reset the si446x radio by applying shutdown and releasing it. After this function si446x_boot should be called. You can check if POR has completed by waiting 4 ms or by polling GPIO 0, 2, or 3. When these GPIOs are high, it is safe to call si446x_boot.
- Return Value:** None
-
- Function Name:** void **si446x_power_up**(uint8_t BOOT_OPTIONS, uint8_t XTAL_OPTIONS, uint32_t XO_FREQ);
- Description:** This function is used to initialize after power-up the radio chip.
- Input Parameter(s):** BOOT_OPTIONS : Patch mode selector
XTAL_OPTIONS : Select if TCXO is in use
XO_FREQ :Frequency of TCXO or external crystal oscillator in Hz
- Return Value:** None
- Note:** Before this function si446x_reset should be called.
-
- Function Name:** uint8_t **si446x_configuration_init**(const uint8_t* pSetPropCmd
- Description:** This function is used to load all properties with a list of NULL terminated set property commands.
- Input Parameter(s):** *pSetPropCmd : First element of the list to be loaded
- Return Value:** SI446X_COMMAND_ERROR : the list of commands is wrong
SI446X_CTS_TIMEOUT : If a CTS error occurs
SI446X_SUCCESS : If loading is successful
- Note:** Before this function si446x_reset should be called.
-
- Function Name:** void **si446x_part_info**(void);
- Description:** This function sends the PART_INFO command to the radio and receives the answer Si446xCmd union.
- Return Value:** None
-
- Function Name:** void **si446x_start_tx**(uint8_t CHANNEL, uint8_t CONDITION, uint16_t TX_LEN);
- Description:** Sends START_TX command to the radio.
- Input Parameter(s):** CHANNEL : Channel number.
CONDITION : Start TX condition.
TX_LEN : Payload length (exclude the PH generated CRC).
- Return Value:** None
-
- Function Name:** void **si446x_start_rx**(uint8_t CHANNEL, uint8_t CONDITION, uint16_t RX_LEN, uint8_t NEXT_STATE1, uint8_t NEXT_STATE2, uint8_t NEXT_STATE3);
- Description:** Sends START_RX command to the radio.

Input Parameter(s):	CHANNEL : Channel number. CONDITION : Start RX condition. RX_LEN : Payload length (exclude the PH generated CRC). NEXT_STATE1 : Next state when Preamble Timeout occurs. NEXT_STATE2 : Next state when a valid packet received. NEXT_STATE3 : Next state when invalid packet received (e.g., CRC error).
Return Value:	None
Function Name:	void si446x_get_int_status (uint8_t PH_CLR_PEND, uint8_t MODEM_CLR_PEND, uint8_t CHIP_CLR_PEND);
Description:	Get the Interrupt status/pending flags form the radio and clear flags if requested.
Input Parameter(s):	PH_CLR_PEND : Packet Handler pending flags clear. MODEM_CLR_PEND : Modem Status pending flags clear. CHIP_CLR_PEND : Chip State pending flags clear.
Return Value:	None
Function Name:	void si446x_gpio_pin_cfg (uint8_t GPIO0, uint8_t GPIO1, uint8_t GPIO2, uint8_t GPIO3, uint8_t NIRQ, uint8_t SDO, uint8_t GEN_CONFIG);
Description:	Send GPIO pin config command to the radio and reads the answer into Si446xCmd union.
Input Parameter(s):	GPIO0 : GPIO0 configuration. GPIO1 : GPIO1 configuration. GPIO2 : GPIO2 configuration. GPIO3 : GPIO3 configuration. NIRQ : NIRQ configuration. SDO : SDO configuration. GEN_CONFIG : General pin configuration.
Return Value:	None
Function Name:	void si446x_set_property (uint8_t GROUP, uint8_t NUM_PROPS, uint8_t START_PROP, ...);
Description:	Send SET_PROPERTY command to the radio.
Input Parameter(s):	GROUP : Property group. NUM_PROPS : Number of property to be set. The properties must be in ascending order in their sub-property aspect. Max. 12 properties can be set in one command. START_PROP : Start sub-property address.
Return Value:	None
Function Name:	void si446x_change_state (uint8_t NEXT_STATE1);
Description:	Issue a change state command to the radio.
Input Parameter(s):	NEXT_STATE1 : Next state
Return Value:	None

9.5.4. Extended Radio Driver

Function Name: void **si446x_nop**(void)
Description: Sends NOP command to the radio. Can be used to maintain SPI communication.
Return Value: None

Function Name: void **si446x_fifo_info**(uint8_t FIFO)
Description: Send the FIFO_INFO command to the radio. Optionally resets the TX/RX FIFO. Reads the radio response back Si446xCmd union.
Input Parameter(s): FIFO : RX/TX FIFO reset flags.
Return Value: None

Function Name: void **si446x_write_tx_fifo**(uint8_t numBytes, uint8_t* pTxData)
Description: The function can be used to load data into TX FIFO.
Input Parameter(s): numBytes : Data length to be load.
pTxData : Pointer to the data (uint8_t*).
Return Value: None

Function Name: void **si446x_read_rx_fifo**(uint8_t numBytes, uint8_t* pRxData)
Description: Reads the RX FIFO content from the radio.
Input Parameter(s): numBytes : Data length to be read.
pRxData : Pointer to the buffer location.
Return Value: None

Function Name: void **si446x_get_property**(uint8_t GROUP, uint8_t NUM_PROPS, uint8_t START_PROP)
Description: Get property values from the radio. Reads them into Si446xCmd union.
Input Parameter(s): GROUP : Property group number.
NUM_PROPS : Number of properties to be read.
START_PROP : Starting sub-property number.
Return Value: None

9.5.5. Full Radio Driver

- Function Name:** void **si446x_func_info**(void)
Description: Sends the FUNC_INFO command to the radio, then reads the response into Si446xCmd union.
Return Value: None
- Function Name:** void **si446x_frr_a_read**(uint8_t respByteCount)
Description: Reads the Fast Response Registers starting with A register into Si446xCmd union.
Input Parameter(s): respByteCount : Number of Fast Response Registers to be read.
Return Value: None
- Function Name:** void **si446x_frr_b_read**(uint8_t respByteCount)
Description: Reads the Fast Response Registers starting with B register into Si446xCmd union.
Input Parameter(s): respByteCount : Number of Fast Response Registers to be read.
Return Value: None
- Function Name:** void **si446x_frr_c_read**(uint8_t respByteCount)
Description: Reads the Fast Response Registers starting with C register into Si446xCmd union.
Input parameter(s): respByteCount : Number of Fast Response Registers to be read.
Return value: None
- Function Name:** void **si446x_frr_d_read**(uint8_t respByteCount)
Description: Reads the Fast Response Registers starting with D register into Si446xCmd union.
Input Parameter(s): respByteCount : Number of Fast Response Registers to be read.
Return Value: None
- Function Name:** void **si446x_get_adc_reading**(uint8_t ADC_EN)
Description: Reads the ADC values from the radio into Si446xCmd union.
Input Parameter(s): ADC_EN : ADC enable parameter.
Return Value: None
- Function Name:** void **si446x_get_packet_info**(uint8_t FIELD_NUMBER_MASK, uint16_t LEN, int16_t DIFF_LEN)
Description: Receives information from the radio of the current packet. Optionally can be used to modify the Packet Handler properties during packet reception.
Input Parameter(s): FIELD_NUMBER_MASK : Packet Field number mask value.
LEN : Length value.
DIFF_LEN : Difference length.
Return Value: None

Function Name: void **si446x_get_ph_status**(uint8_t PH_CLR_PEND)
Description: Gets the Packet Handler status flags. Optionally clears them.
Input Parameter(s): PH_CLR_PEND Flags to clear.
Return Value: None

Function Name: void **si446x_get_modem_status**(uint8_t MODEM_CLR_PEND)
Description: Gets the Modem status flags. Optionally clears them.
Input Parameter(s): MODEM_CLR_PEND : Flags to clear.
Return Value: None

Function Name: void **si446x_get_chip_status**(uint8_t CHIP_CLR_PEND)
Description: Gets the Chip status flags. Optionally clears them.
Input Parameter(s): CHIP_CLR_PEND Flags to clear.
Return Value: None

Function Name: void **si446x_ircal**(uint8_t SEARCHING_STEP_SIZE, uint8_t SEARCHING_RSSI_AVG, uint8_t RX_CHAIN_SETTING1, uint8_t RX_CHAIN_SETTING2)
Description: Performs image rejection calibration. Completion can be monitored by polling CTS or waiting for CHIP_READY interrupt source.
Input Parameter(s): SEARCHING_STEP_SIZE
SEARCHING_RSSI_AVG
RX_CHAIN_SETTING1
RX_CHAIN_SETTING2
Return Value: None

Function Name: void **si446x_protocol_cfg**(uint8_t PROTOCOL)
Description: Sets the chip up for specified protocol.
Input Parameter(s): PROTOCOL:
0 = Packet format is generic, no dynamic reprogramming of packet handler properties.
1 = Packet format is IEEE802.15.4g compliant. The following properties are overridden: PKT_CRC_CONFIG, CRC_ENDIAN/BIT_ORDER in PKT_CONFIG1 for TX and RX, PKT_FIELD_1_CRC_CONFIG for RX.
Other applicable properties in the packet handler group still need to be programmed.
Field 1 should have the length of 16 bits to contain the PHR with PKT_LEN_FIELD_SOURCE set to 1 for RX.
PSDU field shall use Field 2 with variable length.
Field 2 length should be set to the maximum allowed including the anticipated FCS length. It is anticipated that the FCS will be calculated by the host and transmitted over the air. Si4440 will receive PHR and put FCS in the FIFO for the host to retrieve and check. Therefore, CRC shouldn't be enabled on Si4440.
Return Value: None

Function Name: void **si446x_request_device_state**(void)
Description: Requests the current state of the device and lists pending TX and RX requests
Return Value: None

Function Name: void **si446x_rx_hop**(uint8_t INTE, uint8_t FRAC2, uint8_t FRAC1, uint8_t FRAC0, uint8_t VCO_CNT1, uint8_t VCO_CNT0)
Description: While in RX state this will hop to the frequency specified by the parameters and start searching for a preamble.
Input Parameter(s): INTE : New INTE register value.
FRAC2 : New FRAC2 register value.
FRAC1 : New FRAC1 register value.
FRAC0 : New FRAC0 register value.
VCO_CNT1 : New VCO_CNT1 register value.
VCO_CNT0 : New VCO_CNT0 register value.
Return Value: None

Function Name: void **si446x_agc_override**(uint8_t AGC_OVERRIDE)
Description: This command is used to allow override of the AGC.
Input Parameter(s): AGC_OVERRIDE AGC override parameters.
Return Value: None

9.6. Patching the Radio

9.6.1. What is a Radio Patch?

A Radio patch is a piece of software provided by Silicon Labs to change the operation of the radio chip. The patch runs on the internal MCU of the radio chip and fixes bugs or adds/improves features of the firmware in the internal ROM. It is distributed in binary form and has to be loaded in the RAM of the radio chip by the host MCU through the SPI interface. Since the RAM is volatile memory, the information stored in it gets lost when the power of the chip is removed. Consequently, the content of the firmware patch has to be downloaded into the radio chip each time after performing a power on reset.

9.6.2. Patch Usage

In paragraph 8.2 Figure 20 describes the radio wake-up process. To apply a patch, the patch content has to be sent to the radio chip after POR but before issuing the POWER_UP command.

The patch is provided in a text file that has a .csg extension. Following several lines of comments that identify the patch, it contains the patch data lines, eight bytes in a line. Each line has to be sent to the chip as an eight byte long command. A CTS reply has to be read from the chip after each line.

After all lines are sent, the POWER_UP command has to be issued with the PATCH bit of the BOOT_OPTIONS argument set to 1.

POWER_UP Command	7	6	5	4	3	2	1	0
CMD	0x02							
BOOT_OPTIONS	PATCH	0	FUNC [5:0]					
XTAL_OPTIONS	0000000							TCXO
XO_FREQ	XO_FREQ[31:24]							
XO_FREQ	XO_FREQ[23:16]							
XO_FREQ	XO_FREQ[15:8]							
XO_FREQ	XO_FREQ[7:0]							

Figure 33. PATCH bit in the POWER_UP API Command

9.6.3. Patch Support in WDS

If a patch is necessary for a selected radio chip version, WDS will apply it automatically without any user interaction.

When deploying an example project from WDS, it includes the necessary patch commands (Si446X_PATCH_CMDS) in the RADIO_CONFIGURATION_DATA_ARRAY automatically according to the selected chip version. The patch commands themselves are defined in the si446x_patch.h file that is also included in the project automatically. This way the patch will be applied as part of the radio initialization process that is described in paragraph 8.3

10. Example Projects

The general structure of an example can be seen in Figure 34. All the tasks are separated into two groups: the Hardware Initialization part and the Main Process part. The Host MCU related tasks initialize the physical interface between the radio and the controller unit including the SPI lines (SCLK, SDI, SDO, NSEL) and general I/O ports (SDN, NIRQ). After that, the internal timer module is initialized in order to provide precise timing for the handlers. Some example projects do not use handlers due to their simplicity (e.g., the un-modulated carrier or the pseudo random transmission projects). Handlers are for monitoring and changing the state of the WMB peripherals. It is necessary to initialize the required handlers before using them. The radio-related tasks prepare the radio for the communication. The shutdown state may be entered by driving the SDN pin high. When coming out of the shutdown state a power on reset will be initiated along with the internal calibrations. After the POR and the BOOT sections, it is required to initialize the radio with the RF settings. It is highly recommended to use the configuration header file (radio_config.h) generated by the wireless development suite. Manual editing in the header file can cause discrepancies and prevent the radio from working correctly. After the radio RF is initialized, the Main Process has two major tasks to do. It continuously updates the peripheral handlers and processes the user application code. The radio can be controlled from a high level due to the layered, customizable, user-friendly radio driver module.

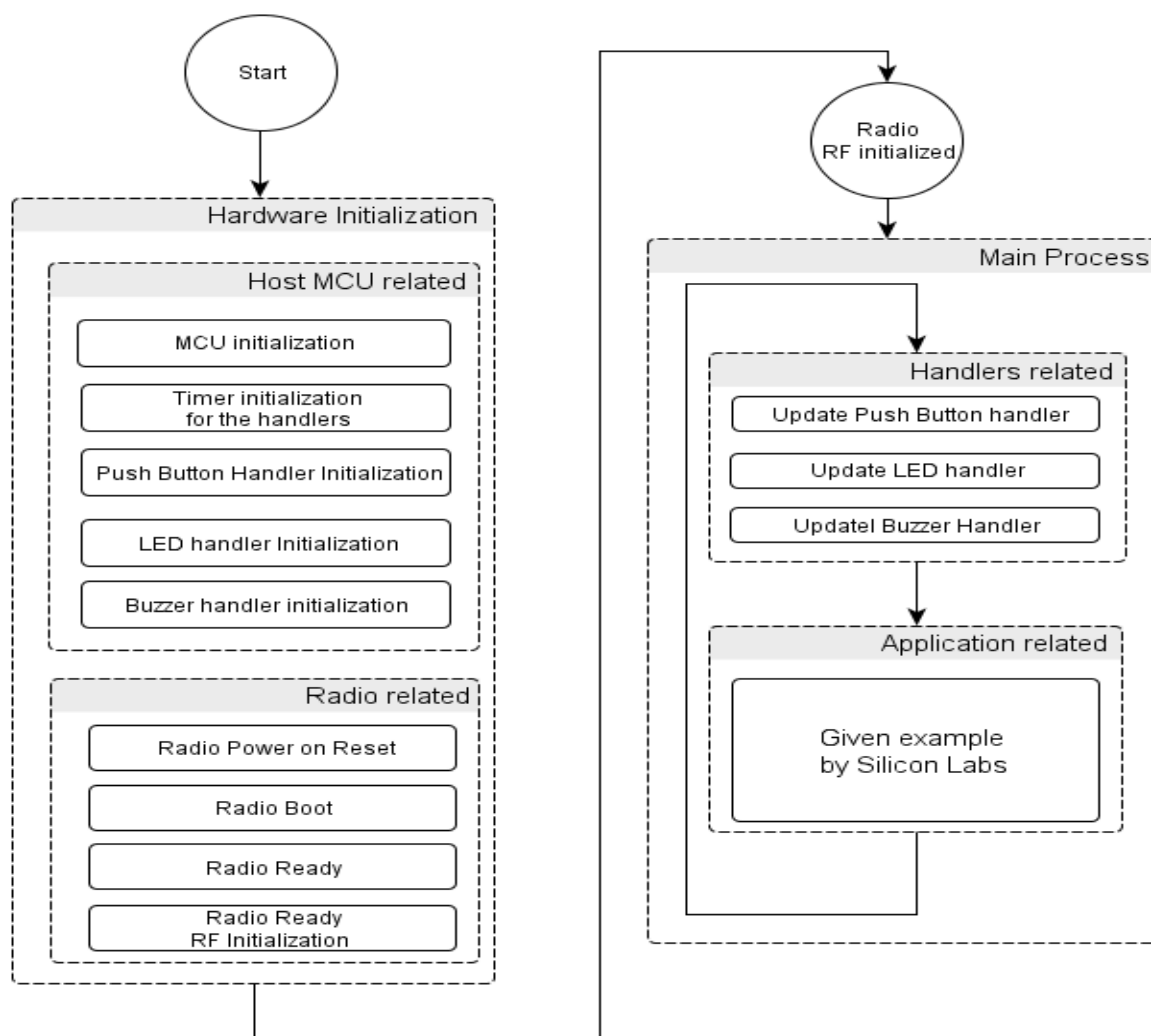


Figure 34. General Structure of an Example Project

All example projects described in this document are configurable for radio parameters and available from within WDS (Wireless Development Suite).

10.1. Un-modulated Carrier (Continuous Wave)

The Radio Configuration Application provides the opportunity to configure the radio to continuous un-modulated transmit mode. It is typically used to measure the output power and the frequency accuracy of the transmitter. The output power and harmonics of the transmitted signal can be easily verified with a spectrum analyzer. You can also customize the RF parameters of the continuous wave transmission. Having initialized the radio with the required RF parameters, the 'vRadio_Start_Tx()' function can start the continuous wave mode.

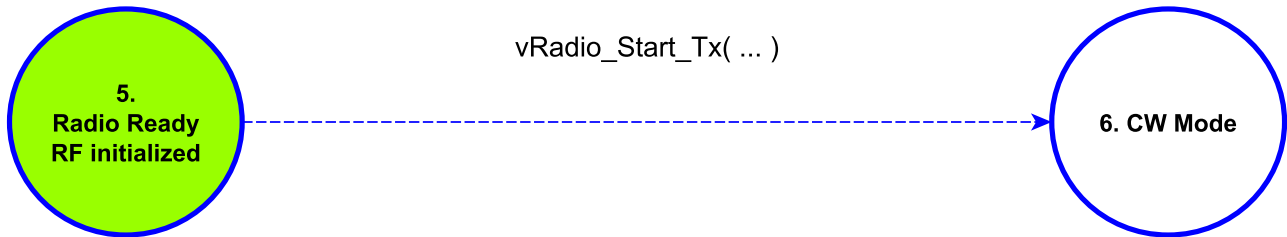


Figure 35. Activate CW Mode

10.2. Pseudo-random Transmission (PN9)

The Radio Configuration Application provides the opportunity to configure the radio to continuous modulated transmit mode. It is typically used to evaluate the transmit parameters of the radio, such as occupied bandwidth and adjacent channel selectivity. The internal PN9 random generator of the radio used to modulate the output power in "PN9" test mode. Having initialized the radio with the required RF parameters, the 'vRadio_Start_Tx()' function can start the PN9 mode.

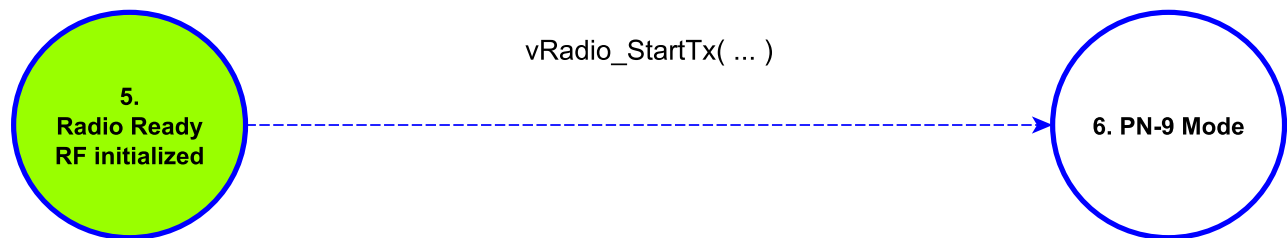


Figure 36. Activate PN9 Mode

10.3. Direct Transmission (Synchronous)

For legacy systems that perform packet handling within the host MCU or other baseband chip, it may not be desirable to use the FIFO. The direct transmission example code bypasses the TX FIFO entirely. The TX modulation data is applied to an input pin of the chip. Data is not stored in the TX_FIFO for transmission at a later time. The host MCU sends data to the radio. Data can be synchronized either with the rising or the falling edge of the clock signal provided by an output pin of the RF chip.

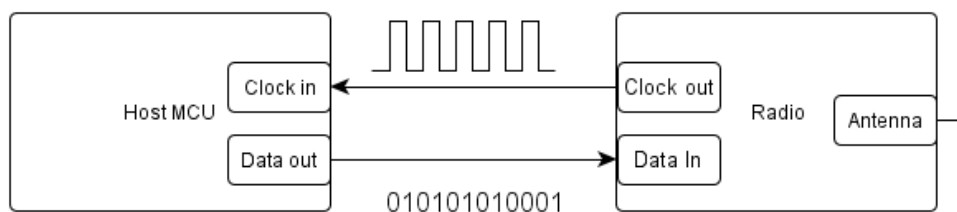


Figure 37. GPIO Connections Between the Radio and the Host MCU

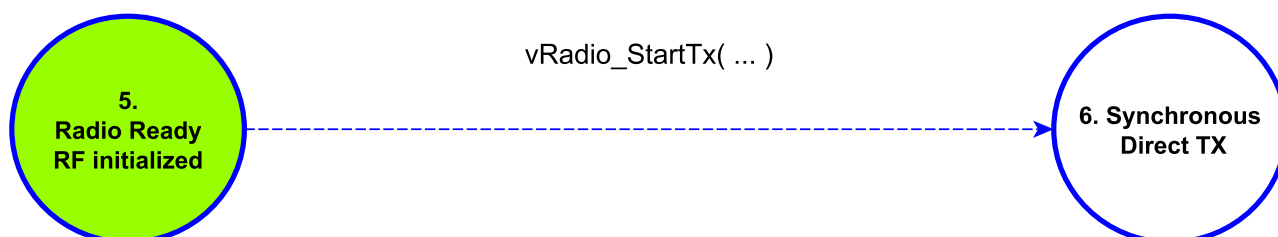


Figure 38. Activate Direct Synchronous Transmission

10.4. Direct Reception (Synchronous and Asynchronous)

For legacy systems that perform packet handling within the host MCU or other baseband chip, it may not be desirable to use the FIFO. The direct reception example code bypasses the RX FIFO entirely. The RX modulation data is provided by an output pin of the RF chip to the host MCU. Data is not stored in the RX_FIFO after the reception. In synchronous mode, the data and the clock signals are synchronized.

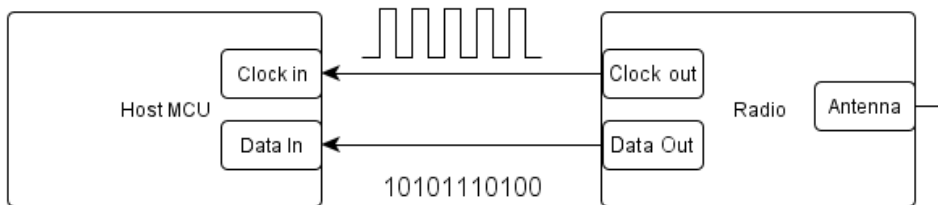


Figure 39. GPIO connections between the radio and the host MCU

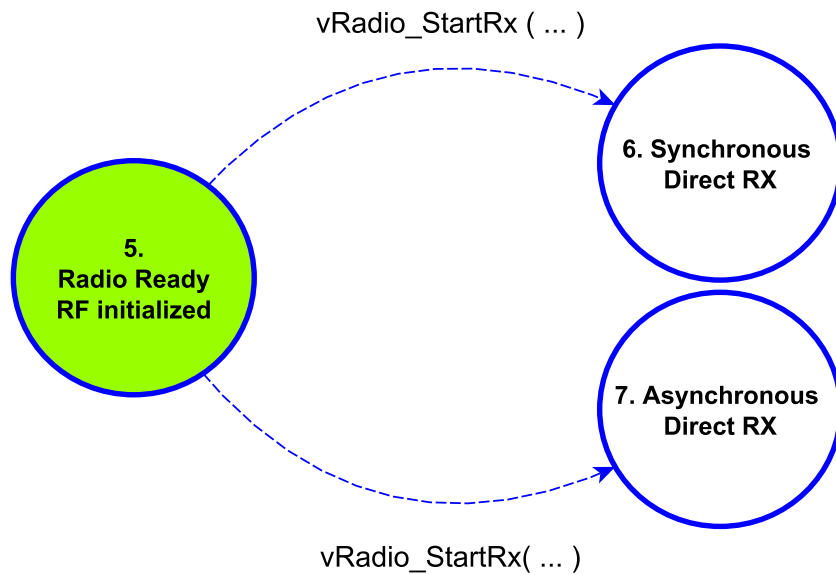


Figure 40. Activate Direct Synchronous or Asynchronous Reception

10.5. Standard Packet Transmission

The purpose of the standard packet transmission example code is to demonstrate how the radio can send packets in FIFO mode. If any of the four push buttons are pressed on the wireless motherboard then the host MCU will load the appropriate packet content in TX_FIFO and after that will send it. The payload is a pre-defined content, namely "BUTTONx" where x can be 1, 2, 3, or 4. The used packet can be seen below:

Table 10. Structure of the Standard Packet

Preamble	Sync Word	Payload	CRC
32-40 bits	2 byte	N* byte	2 byte

*Note: $N \leq 62$ byte

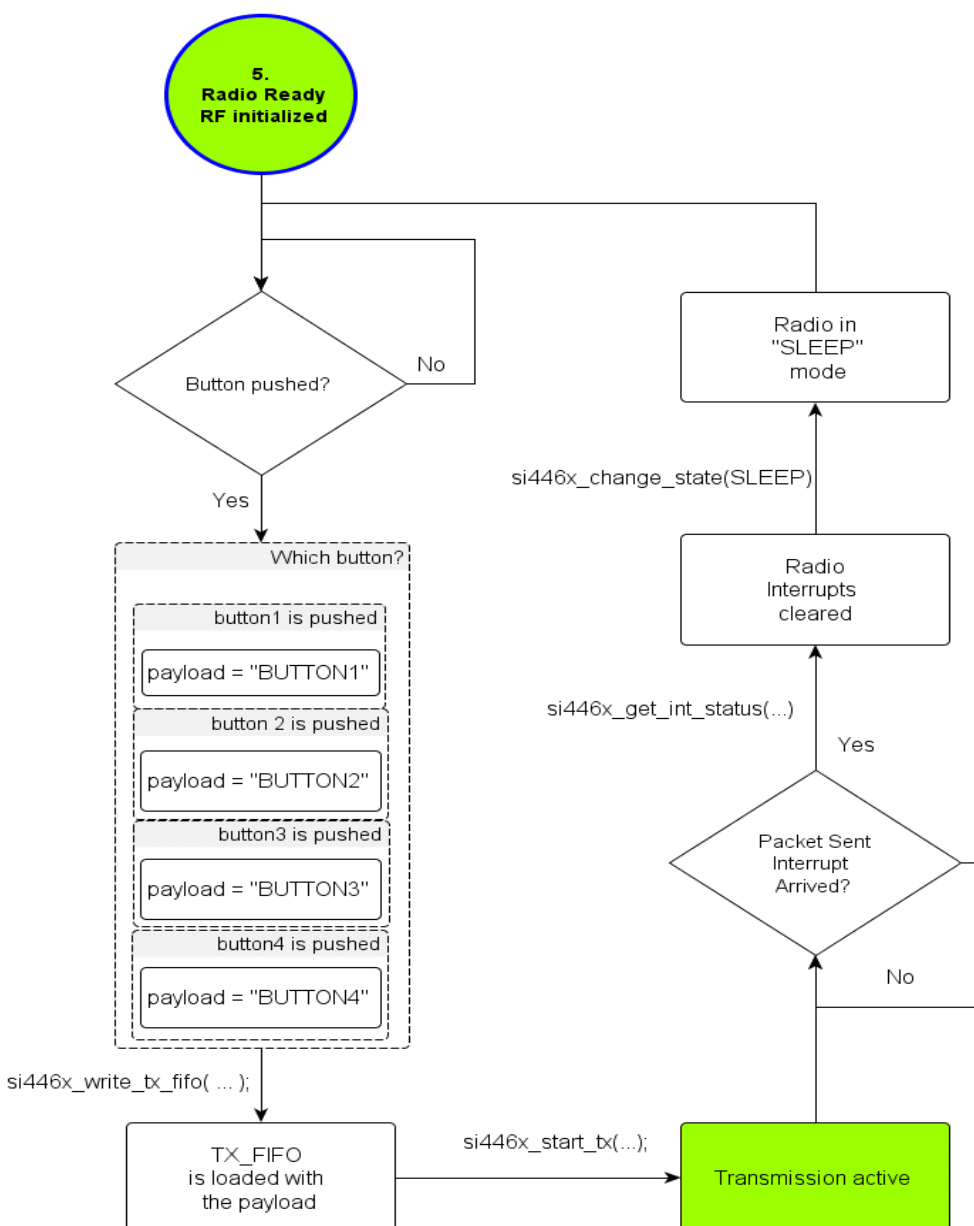


Figure 41. Transmission Flowchart

10.6. Standard Packet Reception

The purpose of the standard packet reception example code is to demonstrate how the radio can receive packets in FIFO mode. Additionally, the receivers and the transceivers are compatible with Si4010 transmitters used in other development kits. Therefore, it supports several packet types:

- Fix packet content
- Key Fob packet (type A)
- Key Fob packet (type B)

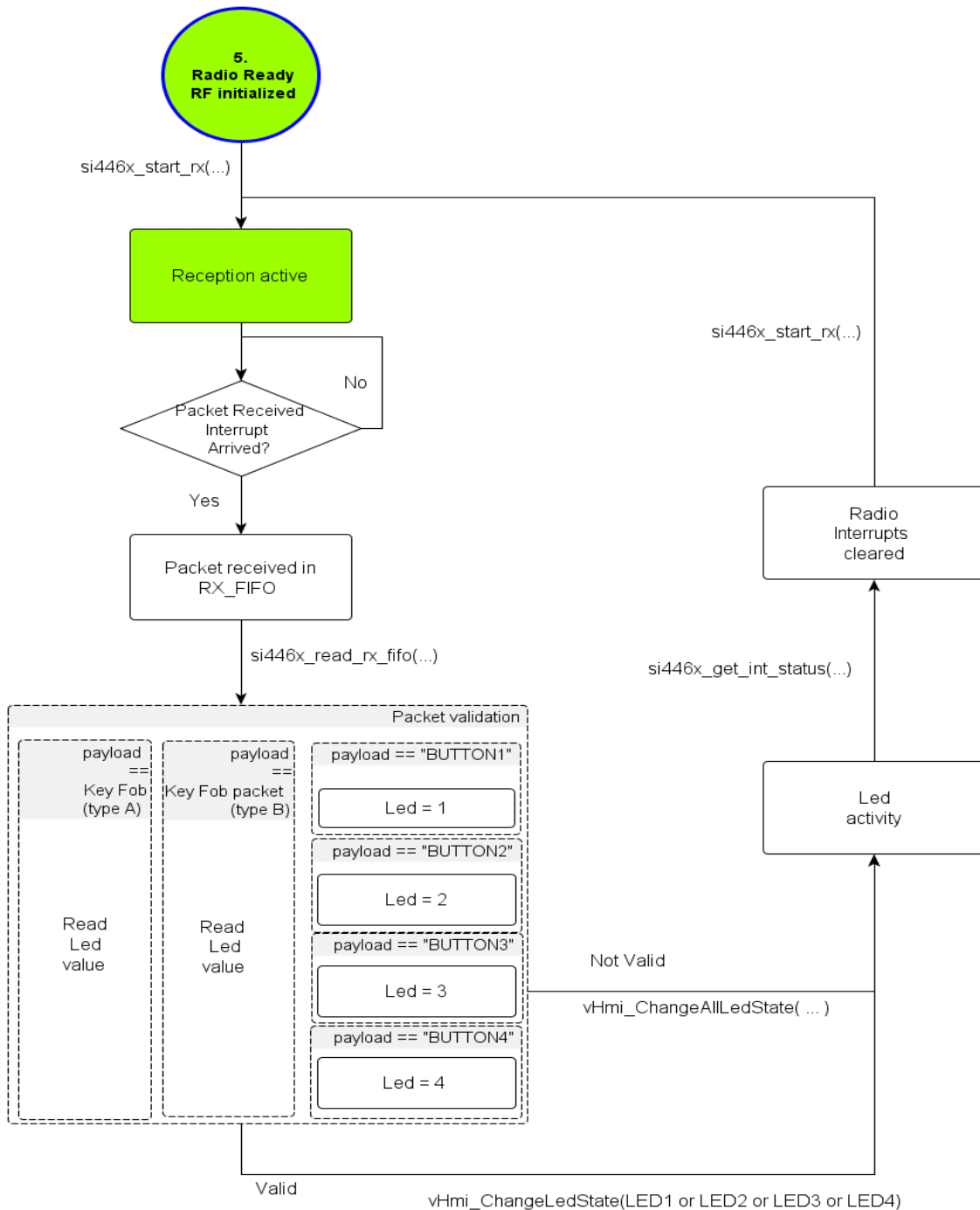


Figure 42. Reception Flowchart

10.6.1. Packet with Fixed Content

The payload has its pre-defined content, namely “BUTTONx” where x can be 1, 2, 3, or 4. The used packet can be seen below:

Table 11. Structure of the Packet with Fixed Content

Preamble	Sync Word	Payload	CRC
32–40 bits	2 byte	N byte	byte

10.6.2. Key Fob Packet (Type A)

The following development kits use also this type of packet:

- Si4010 Key Fob Demo Kit with AES Capability 434 MHz
- Si4010 Key Fob Demo Kit with AES Capability 868 MHz
- Si4010 Simplified Key Fob Demo Kit 434 MHz
- Si4010 Simplified Key Fob Demo Kit 868 MHz

The used packet is shown in Table 12.

Table 12. Structure of the Key Fob Packet (Type A)

Preamble	Sync Word	Function Control Byte	One's Complement of Function Byte	Function Control Byte
4 byte	2 byte	1 byte	1 byte	1 byte

Table 13. The Structure of the Function Control Byte:

OUT3 F1	OUT3 F0	OUT2 F1	OUT2 F0	OUT1 F1	OUT1 F0	OUT0 F1	OUT0 F0
------------	------------	------------	------------	------------	------------	------------	------------

OUT3/F1 sent out first, OUT0–OUT3 represent the four LED outputs, F0–F1 represents the function bits. Output functions are controlled by the function bits. Function bits' operations are shown on the following table:

F1	F0	Function
0	0	No change
0	1	Sets output logical low (LED is OFF)
1	0	Sets output logical high (LED is ON)

10.6.3. Key Fob Packet (Type B)

The following development kits use also this type of packet:

- Si4010/Si4355 Key Fob Development Kit
- Si4010/Si4355 One-Way AES Development Kit
- Si4010/Si4355 One-Way Sub-GHz Key Fob to LED Receiver Stick
- Si4455 Two-Way Sub-GHz Key Fob to LED Receiver Stick

The used packet can be seen below:

Table 14. Structure of the Key Fob Packet (Type B)

Preamble	Sync Word	Chip ID	Status	Packet Count	CRC
13 byte	2 byte	4 byte	1 byte	2 byte	2 byte

Table 15. More Detailed Structure of Key Fob Packet (Type B)

Number of Bytes	Field Name	Description
13	Preamble	0xAA
2	Sync	0x2D, 0xD4
4	Chip Id	Unique, factory-burned chip ID
1	Status	Lower 5 bits are the button information
2	Packet Count	Rolling counter
2	CRC-16	Generator $X^{16}+X^{15}+X^2+1$, start value 0xFFFF

10.7. Custom Packet Transmission

This example code works the same way as the standard packet transmission with the exception that the content of the payload field in the packet is customizable.

10.8. Custom Packet Reception

This example code works the same way as the standard packet transmission with the exception that the content of the payload field in the packet is customizable.

10.9. Two-Way Packet (Bidirectional Packet)

The purpose of the two-way packet example code is to demonstrate how to use the radio for a two-way link communication. By default, the two devices are in receiver mode and waiting for either a request message or a pushbutton event from the user. When the first pushbutton is pushed on one of the WMBs, it sends a request immediately to the other radio. The LED activity shows if the radio packet is sent successfully. The receiver replies with an acknowledgment followed by LED activity.

The payload of the request is 64 bytes long and customizable as well.

Table 16. Structure of the Request Message

Preamble	Sync Word	Payload	CRC
32–40 bits	2 bytes	64 bytes	2 byte

The payload of the reply is the 3 byte long “ACK” string.

Table 17. Structure of the Reply Message

PREAMBLE	SYNC WORD	Payload	CRC
32–40 bits	2 bytes	3 bytes	2 bytes

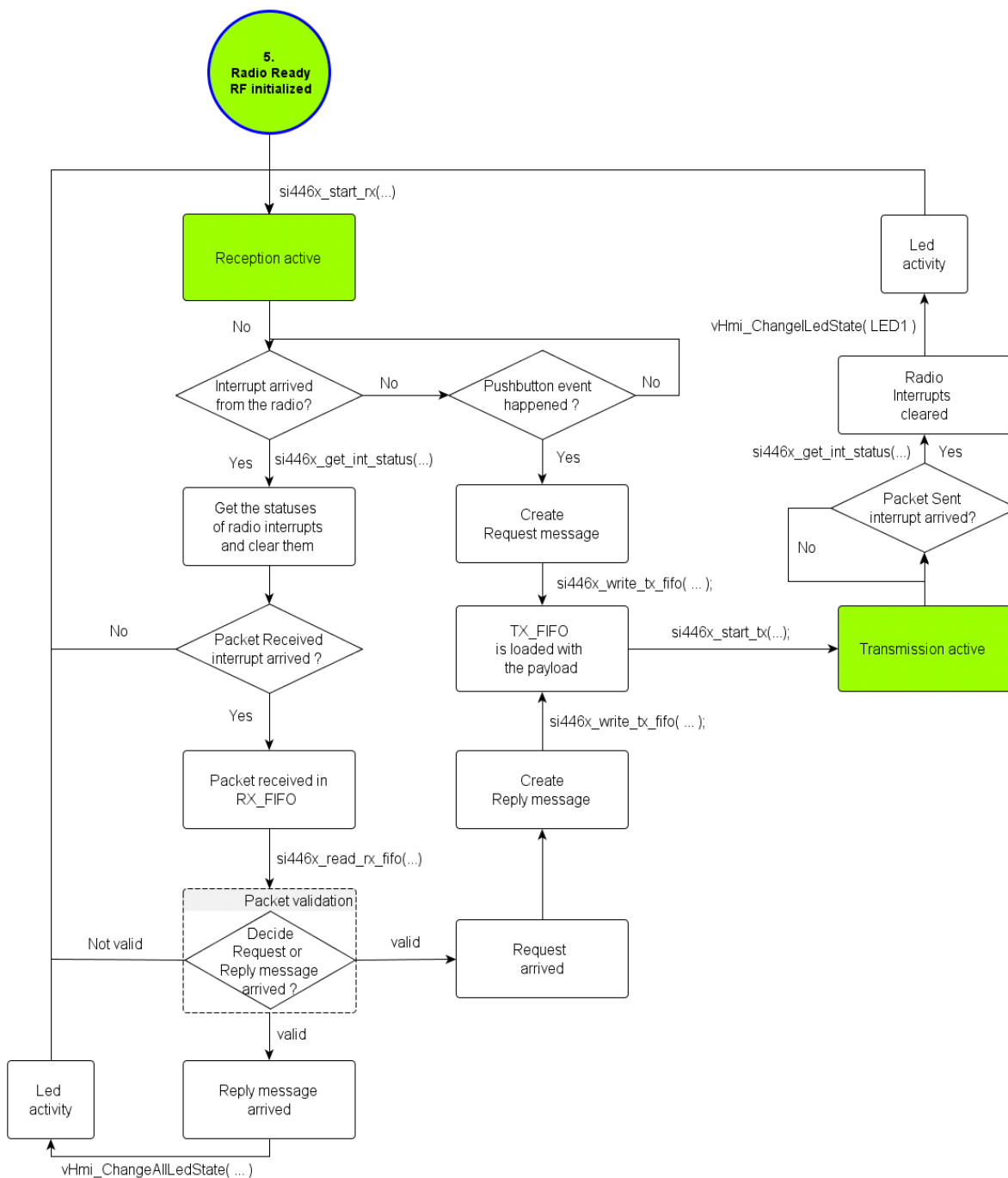


Figure 43. Two-way Packet Flowchart

10.10. 802.15.4g Bidirectional Project

This example project demonstrates how to use the Si446x radio IC for 15.4g packets in regular boot mode. Just like the standard Bidirectional code (Section 10.9 Two-Way Packet), it implements a two-way communication link. Pushing a button on one side results in sending a packet to the other side; the incoming packet is acknowledged by sending back an ACK packet (see Figure 44 below). A node can be either a sender or a receiver. The main difference between the two projects is that the 15.4g example shows how to configure the device to meet the 802.15.4g SUN-PHY specification, which is described by the IEEE 802.15.4g-2012 standard (referred to as “standard” later in this document). Although the standard specifies both the PHY and the MAC layers, only the PHY has been implemented when designing the example project (from a radio IC configuration standpoint the PHY layer is the point of interest; the MAC is one layer above the PHY and needs to be implemented by the user application. There is no HW/API support in regular boot mode for the MAC).

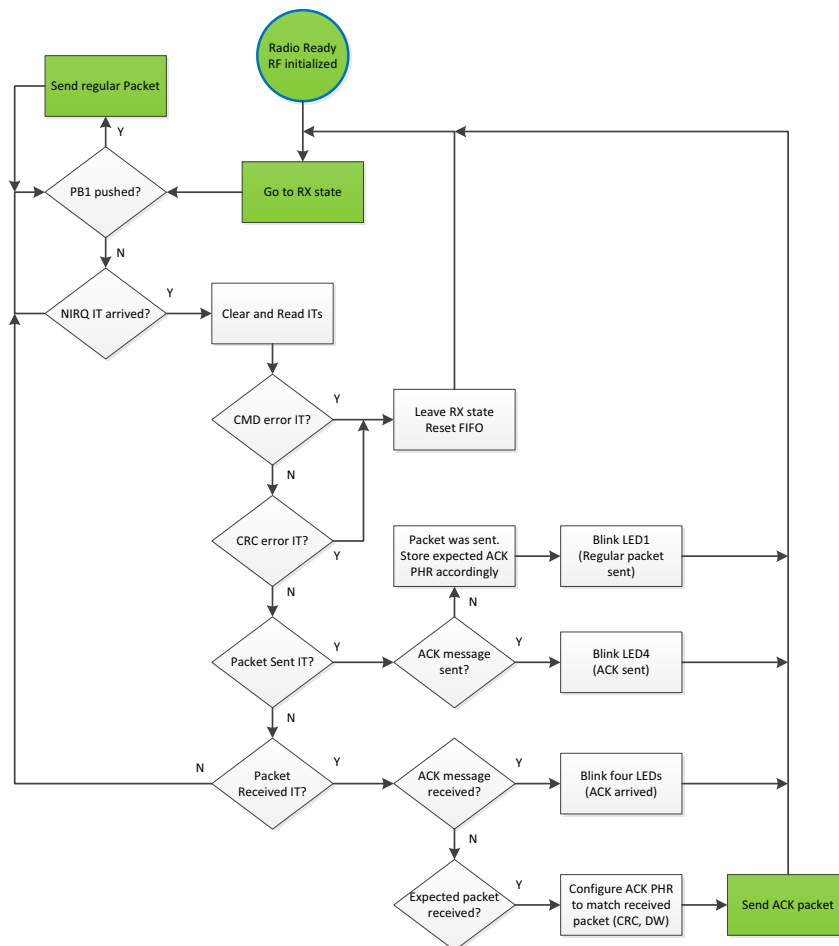


Figure 44. 802.15.4g Bidirectional Example Code Functional Flowchart

Some of the controls in WDS are limited or defaulted to certain values in order to have the right radio configuration for the 15.4g example project. For example, only 2GFSK and 4GFSK modulation types, certain frequency bands, and data rates are available. Also, preamble, sync word patterns, CRC, and data whitening configurations are all fixed on the WDS GUI. The CRC and data whitening polynomials, seeds, bit order, etc, are all specified by the standard and have to be configured carefully. Due to the standard requirements, the variable packet length feature has to be enabled for the RX side.

From a radio packet handler perspective, two fields are used. Field 1 for the PHR (packet header), Field 2 for the

PSDU (payload + CRC). The PHR contains fields to select between 2- and 4-byte CRCs (FCS field), data whitening enable/disable (DW field), and length (Length field). The mode switching field (MS field) is disabled (not supported by the radio). The packet feature and the naming convention of the standard and the radio IC are shown in Figure 45.

							MAC header (MHR)	MAC payload	MAC footer (MFR)
PPDU (PHY Protocol Data Unit)									
SHR (Synchronization header)		PHR (PHY header)					PSDU (PHY Service Data Unit or PHY payload)		
SHR	SHR	Bit 0	1-2	3	4	5-15			
Preamble	Sync field	MS	R1-R0	FCS type	DW	L10-L0	Payload		FCS
Preamble	Sync word	Field 1					Field 2		

Figure 45. Packet Structure and Naming Convention (802.15.4g–blue, Si4468–orange)

The Si446x supports automatic PHR parsing (see GLOBAL_CONFIG:PROTOCOL in the API) which makes it possible to receive packets sent in different ways. Once configured correctly and in receive mode, the radio may receive any packet regardless of how it was sent: with 2- or 4-byte CRC, with or without data whitening, and without initially knowing how many bytes to expect. It recognizes the FCS, DW, and Length fields of the PHR, and based on that information, it runs the built-in, 16- or 32-bit built-in CRC engine, enables or disables data whitening, and receives the right amount of bytes. A given packet can be sent in one of four ways:

- 16-bit CRC, DW disabled
- 16-bit CRC, DW enabled
- 32-bit CRC, DW disabled
- 32-bit CRC, DW enabled

No matter which of these four options is used to send a packet in the example project, the RX side will receive the packet, and send back the ACK accordingly. That is to say if a project is configured to send a packet with 16-bit CRC, DW disabled, it is still able to receive a packet with 32-bit CRC, DW enabled and blinks the LED as long as the payload content is correct. The PHR can be different, but the payload has to be the same. The ACK that it sends back will match that of the sender, 32-bit CRC, DW enabled.

To support the two FCSs, CRC-16 and CRC-32, there are two built-in CRC engines in the radio IC that are running in parallel (referred to as Primary and Alternative CRC in the code). The Alternative is used for 32-bit, the Primary for the 16-bit CRC. The PHR parser will pick the right one to calculate the CRC.

On the transmit side there is no automatic parsing. The PHR field has to be entered manually into the TX FIFO, and the user application has to select the right CRC, enable/disable data whitening accordingly. Note that the CRC and data whitening apply only to the PSDU payload which excludes the PHR. Due to this, two fields are configured: Field 1 for the PHR with no CRC and DW, and Field 2 for the payload + CRC with optional DW.

AN633

The standard requires the data bytes being sent in an LSB manner, which is taken care of by the radio IC. The data, i.e., PHR + payload, is being put into the TX FIFO in MSB, but sent over the air in LSB. On the receive side, the payload is being read out from the RX FIFO in MSB, but the PHR will be in LSB (!). Taking an example from the standard where 0x40 00 56 is being sent over the air (i.e., 0x02 00 6A in MSB), the four different scenarios are the following:

1. **3-byte payload, CRC-16, DW disabled**

Data to the TX FIFO: | 08 A0 | 02 00 6A | (MSB)
PPDU*: | 55 55 55 55 | 90 4E | 10 05 | 40 00 56 | 27 9E | (LSB transmitted first)
Data from the RX FIFO: | 10 05 | 02 00 6A |

2. **3-byte payload, CRC-16, DW enabled**

Data to the TX FIFO: | 18 A0 | 02 00 6A | (MSB)
PPDU*: | 55 55 55 55 | 90 4E | 18 05 | 4F 70 E5 | 48 DD | (LSB transmitted first)
Data from the RX FIFO: | 18 05 | 02 00 6A |

3. **3-byte payload, CRC-32, DW disabled**

Data to the TX FIFO: | 00 E0 | 02 00 6A | MSB
PPDU*: | 55 55 55 55 | 90 4E | 00 07 | 40 00 56 | 5D 29 FA 28 | (LSB transmitted first)
Data from the RX FIFO: | 00 07 | 02 00 6A |

4. **3-byte payload, CRC-32, DW enabled**

Data to the TX FIFO: | 10 E0 | 02 00 6A | (MSB)
PPDU*: | 55 55 55 55 | 90 4E | 08 07 | 4F 70 E5 | 32 6A 62 60 | (LSB transmitted first)
Data from the RX FIFO: | 08 07 | 02 00 6A |

*Data stream sent over the air. PPDU=SHR + PHR + PSDU.

The preamble pattern is a fixed, 0101 pattern, but can be of any length. The sync word is also fixed, a two-byte, 0x904E pattern (these are the default values in the WDS GUI for the project). The standard allows a maximum of 2047 bytes in length (including the FCS), but the sample code has a limit of 64-bytes. If needed, it can be extended up to the maximum by using TX FIFO Almost Empty, RX FIFO Almost Full Interrupts. For such long packets, please refer to Long Packet TX/RX example projects.

10.11. Custom Packet Reception Containing Variable Length Field

This example code works the same way as the standard packet reception with the exception that the content of the payload fields in the packet is customizable. It is possible to receive a packet whose payload data field length is not known in advance. In such a scenario, the transmitted packet must contain byte(s) that specify the length of the variable field. There is no requirement for length byte(s) to occur at one mandatory location in the packet (e.g., immediately following the Sync Word). However, the receiver must obviously know in advance which byte(s) of the received packet represents the length value. Furthermore, these length byte(s) must be located in the packet before the variable length field, else the packet handler in the receiver will not have the information in time to allocate the received data bytes to the appropriate data field. The variable packet length byte(s) can be either in field1, field2, field3, or field4. The variable packet length information must necessarily be a fixed length field.

The variable-length field need not immediately follow the length byte(s) but must occur after the field containing the length byte(s). The variable-length field can be either in field2, field3, field4, or field5.

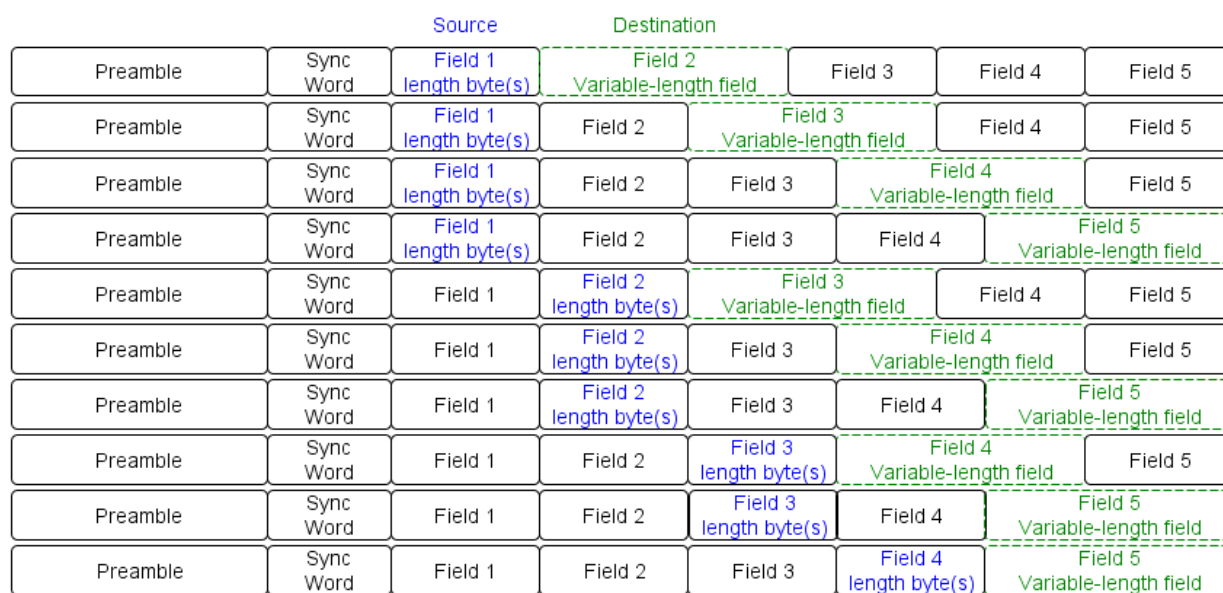


Figure 46. Structure of Valid Variable-Length Packet

		Destination	Source			
Preamble	Sync Word	Field 1 Variable-length field	Field 2 length byte(s)	Field 3	Field 4	Field 5
Preamble	Sync Word	Field 1 Variable-length field	Field 2	Field 3 length byte(s)	Field 4	Field 5
Preamble	Sync Word	Field 1 Variable-length field	Field 2	Field 3	Field 4 length byte(s)	Field 5
Preamble	Sync Word	Field 1 Variable-length field	Field 2	Field 3	Field 4	Field 5 length byte(s)
Preamble	Sync Word	Field 1	Field 2 Variable-length field	Field 3 length byte(s)	Field 4	Field 5
Preamble	Sync Word	Field 1	Field 2 Variable-length field	Field 3	Field 4 length byte(s)	Field 5
Preamble	Sync Word	Field 1	Field 2 Variable-length field	Field 3	Field 4	Field 5 length byte(s)
Preamble	Sync Word	Field 1	Field 2	Field 3 Variable-length field	Field 4 length byte(s)	Field 5
Preamble	Sync Word	Field 1	Field 2	Field 3 Variable-length field	Field 4	Field 5 length byte(s)
Preamble	Sync Word	Field 1	Field 2	Field 3	Field 4 Variable-length field	Field 5 length byte(s)

Figure 47. Structure of Invalid Variable-Length Packet

The field containing the variable packet length information can be 1 byte or 2 bytes long. Length field can be configured as either little endian, where the least significant byte occurs first, or big endian, where the most significant byte occurs first. During the packet reception, the packet handler can be configured so that length field will be stored into the RX_FIFO or stripped off and not stored in the RX_FIFO.

In order to evaluate the capability of receiving variable length packets on the receive side, the appropriate payload has to be constructed on the transmit side. Note that both the source and the destination fields have to be specified with relevant values, and they also have to be located in the packet to be sent. The source field must contain the length information and the destination field must contain as many bytes as the source field strictly specifies. Prior to sending, the constructed packet must be written to the tx fifo.

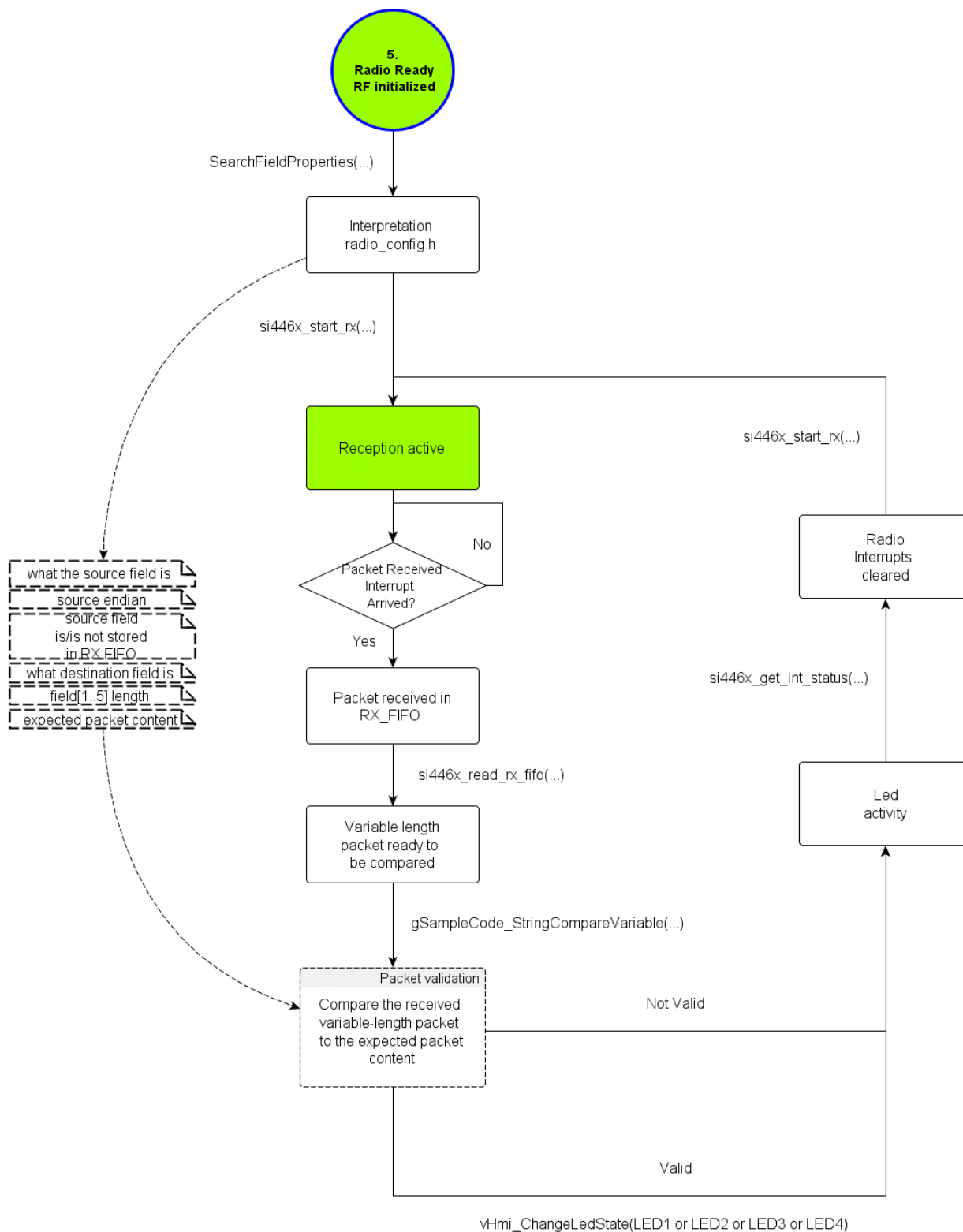


Figure 48. Variable Length Packet Reception Flowchart

10.12. Custom Packet Reception with Matching Capability

The purpose of the packet match reception example code is to demonstrate how the radio can filter packets that are intended for one receiver. The Radio ICs of Si4x6x family provide for fully configurable matching functionality on up to 4 data bytes in the data fields (field1 to field5) of the packet. The match function is typically used to implement header check or broadcast check capability that is used to quickly determine if a packet is intended for one receiver node in a network. If a match is not found, the chip can abort reception of the packet and proceed to scanning for the next packet.

There are four match bytes at configurable locations in the packet. They must be located within the first 32 bytes following the end of the Sync Word. It is also necessary that the offset of the match bytes be in ascending order. The offset location of match #1 byte must be less than the location of match #2 byte, and so on. No two match bytes may have the same offset location. It is not possible to configure two different match functions against the same received data byte.

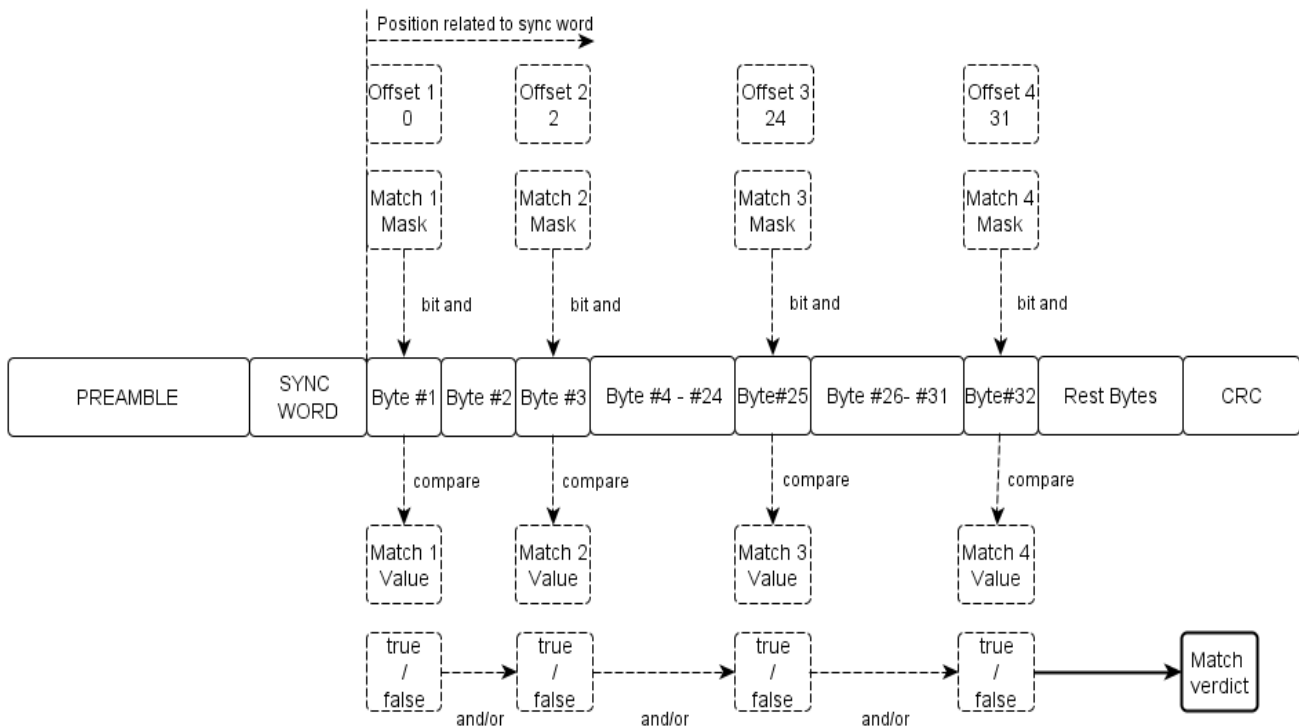


Figure 49. Packet Match Working Mechanism

Once the location of each match byte has been specified, it is next necessary to configure their mask values. A mask value represents a bit pattern which is logically and'ed (bit-wise) with the corresponding received match byte. These mask values are specified in the mask registers' range from 1 to 4. The result is then compared with the corresponding match value. For each successive match byte, it is possible to *logically-or* or *logically-and* its result with the cumulative logical result from the previous matches.

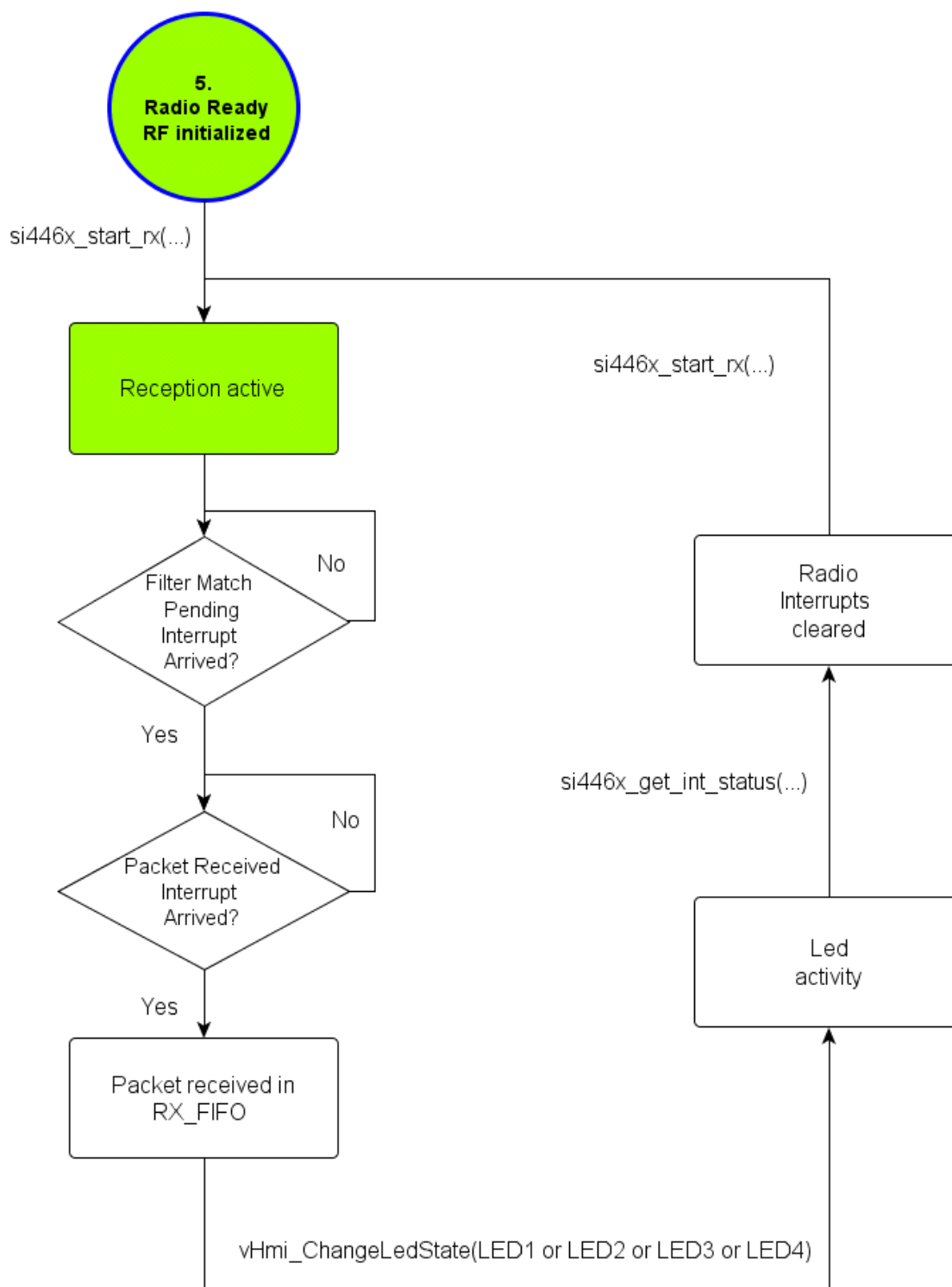


Figure 50. Packet Matching Reception Flowchart

10.13. Packet Reception with Automatic Hopping Capability

The purpose of the standard packet reception with automatic hopping feature example code is to demonstrate how the radio can receive packets in FIFO mode and how rapidly it scans frequency channels to search for signals. Once the device is configured into the RX state, it automatically starts hopping through the pre-configured channels on different frequencies. The given mathematical formula represents how the internal firmware calculates new frequencies to hop.

$$\text{Scannable RF Frequency [Mhz]} = \text{Base Frequency [Mhz]} + \text{Channel_Number} \times \text{Stepsize [Mhz]}$$

The hop table can hold up to 64 channel numbers. The receiver starts receiving at the base channel and hops in sequence from the top of the hop table to the bottom. It stays on that particular channel for receiving a packet. If no receiving packet arrives upon given conditions, it will calculate a new scannable RF frequency. The table will wrap around to the base channel once it reaches the end of the table. A channel number is configured to 0xFF in the table to indicate that the channel should be skipped.

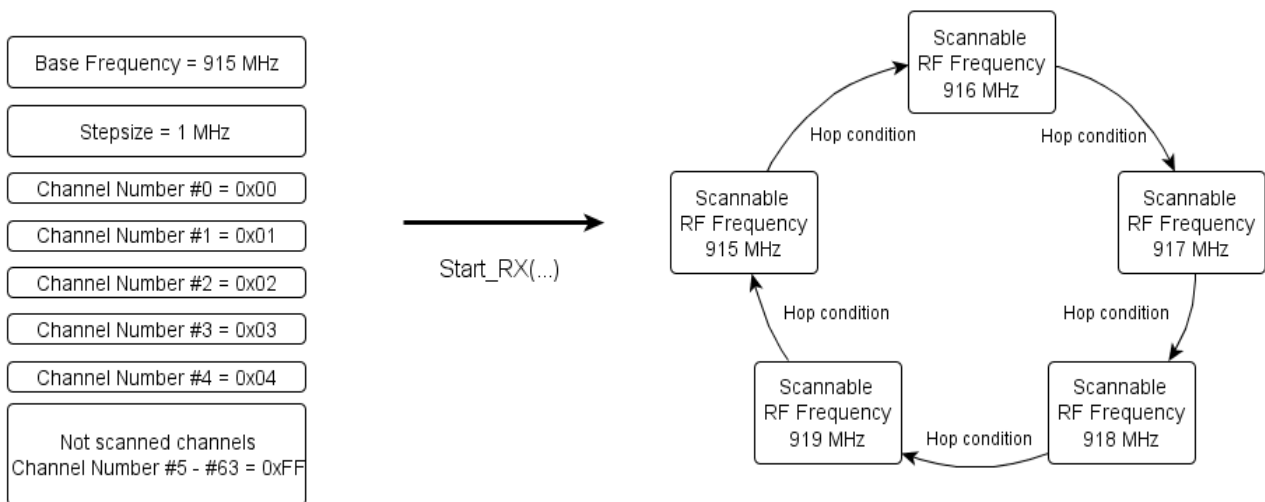


Figure 51. General Working Mechanism of the Automatic RX Hopping

It can be observed on the radio's GPIOs how the radio hops from one channel to another according to the hop condition. The yellow signal shows that it toggles between output high and low when hop occurs. The blue signal shows that it toggles between output high and low when hop table wraps.

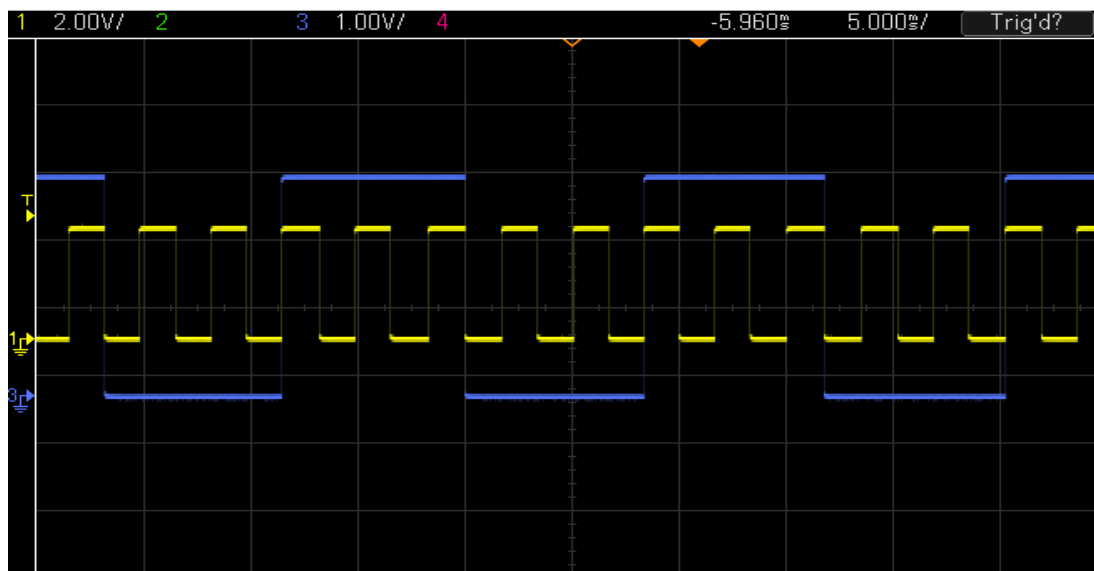


Figure 52. Observation of the Automatic RX Hopping

There are three configurable hop conditions that can be used to determine whether to continue hopping or to stay on a particular channel. These conditions are as follows:

- **RSSI timeout:** After the receiver chain is settled, a timer is started and the RSSI is averaged within this time (if the average is enabled). If the RSSI is above the desired threshold, then the radio stays on the actual channel. If timeout occurs without the RSSI exceeding the threshold, the radio will automatically hop to the next channel. The RSSI timeout can be set in bit units.
- **Preamble timeout (invalid preamble pattern):** Invalid preamble timeout starts once the receiver chain is settled. If it timeouts without detecting the preamble (the preamble detection condition is determined by the preamble detection threshold), then the radio automatically hops to the next channel. The invalid preamble timeout is configurable.

Note: 1. The invalid preamble timeout must be longer than the preamble detection threshold.
2. If the preamble is detected within the timeout and the radio stays on the actual channel, then the sync word conditions determine the further radio behavior.

- **Sync word invalid timeout (invalid or no sync word detected after the preamble):** After the preamble is detected, the radio is looking for the sync word. Once non-preamble bits are received, the radio starts a timer. If the sync word is received within sync word length + 4 bits time, then the radio starts to fill the payload into the RX_FIFO. If the timer timeouts without detecting the sync word, then the radio will hop to the next channel automatically. The sync timeout is not configurable, and it cannot be disabled.

Some considerations:

The RSSI detection is the fastest way to detect if a channel is occupied or not. However this method does not qualify if the received signal is sent with the expected radio parameters; therefore, even an un-modulated disturber signal can influence the RSSI measurement and result in a false detection. Please also note that the read RSSI value is radio configuration dependent; therefore, it is recommended that one measure the input power vs. RSSI curve for each radio settings respectively. The RSSI curve gets more accurate a few dB above the sensitivity level; therefore, it is highly recommended that one use at least a 4 bits time average and set the RSSI threshold to 3dB above the sensitivity level.

The preamble detection method is more reliable than the RSSI method, because it requires demodulating the incoming signal and it acts only on the data stream that is sent with the desired radio parameters. Occurrence of a short preamble pattern is frequently from the white noise that is generated by the demodulator if nothing is transmitted. Therefore, if the preamble detection threshold is set relatively short, then a false preamble detection may occur and prevent the channel hop as long as a next condition does not trigger a hop. This time period may prevent the reception of a valid packet that is transmitted during the same time. To avoid such situations, a longer (1.5 ... 2 bytes) detection threshold is suggested.

AN633

After the radio is configured to the receiver state, the PLL will be settled within 50 μs, followed by the RX settling time. Depending on whether the AFC is used or not, the RX settling time can last an extra 8 or 16 bits time. If the RSSI detection method is used, please consider the RSSI settling time which is an additional 4 bits time. If the frequency hopping system is not time synchronized, then the transmitter can send a packet occasionally while the receiver is continuously scanning the channels. The best approach is to transmit the preamble as long as it takes to scan all channels. That ensures that the receiver will find the preamble and will be able to receive the packet independently on whichever channel it is transmitted. The required minimum preamble length can be calculated as follows:

1. If the RSSI timeout method is used:

$$\min_{\text{bit}}(\text{preamble length}) = [(\text{PLL_settling_time} \times \text{data_rate}) + \text{RX_settling_time} + \text{RSSI_settling_time} + \text{RSSI_timeout}] \times \text{Number_of_Channels_to_scan}$$

2. If the Preamble timeout method is used:

$$\min_{\text{bit}}(\text{preamble length}) = [(\text{PLL_settling_time} \times \text{data_rate}) + \text{RX_settling_time} + \text{Preamble_timeout}] \times \text{Number_of_Channels_to_scan}$$

This following figure shows the receiver timing behavior if both the preamble and the RSSI conditions are used.

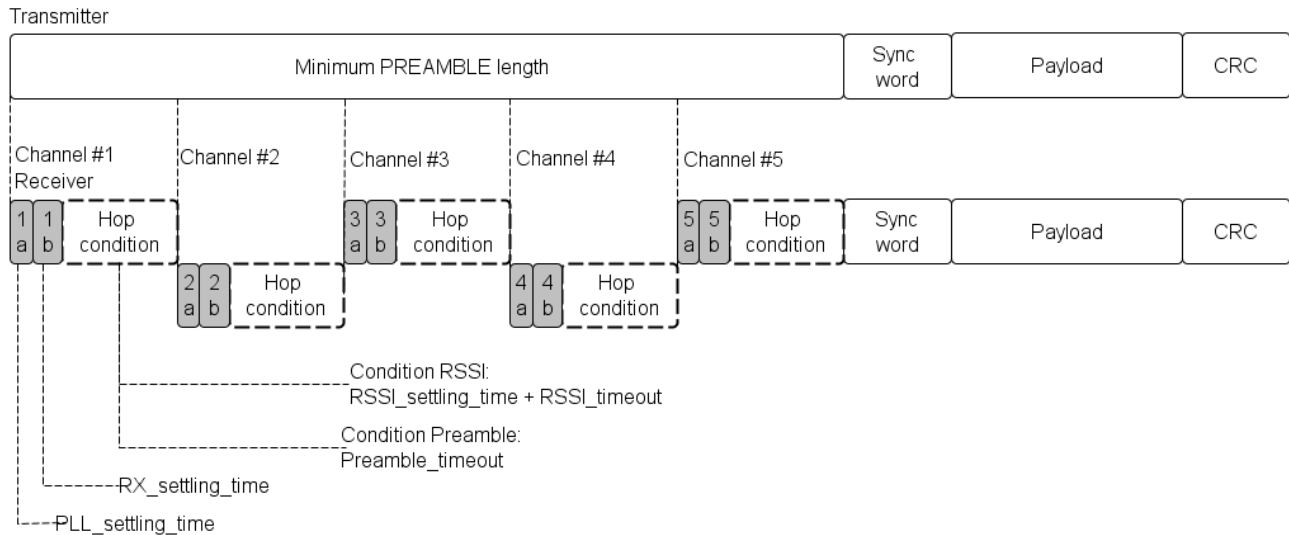


Figure 53. Calculation of the Minimum Preamble Length

3. If the RSSI timeout, the preamble timeout, and the sync word invalid timeout methods are used:

$$\min_{\text{bit}}(\text{preamble length}) = [(\text{PLL_settling_time} \times \text{data_rate}) + \text{RX_settling_time} + \text{RSSI_settling_time} + \text{RSSI_timeout} + \text{Preamble_timeout} + \text{Invalid_sync_timeout}] \times \text{Number_of_Channels_to_scan}$$

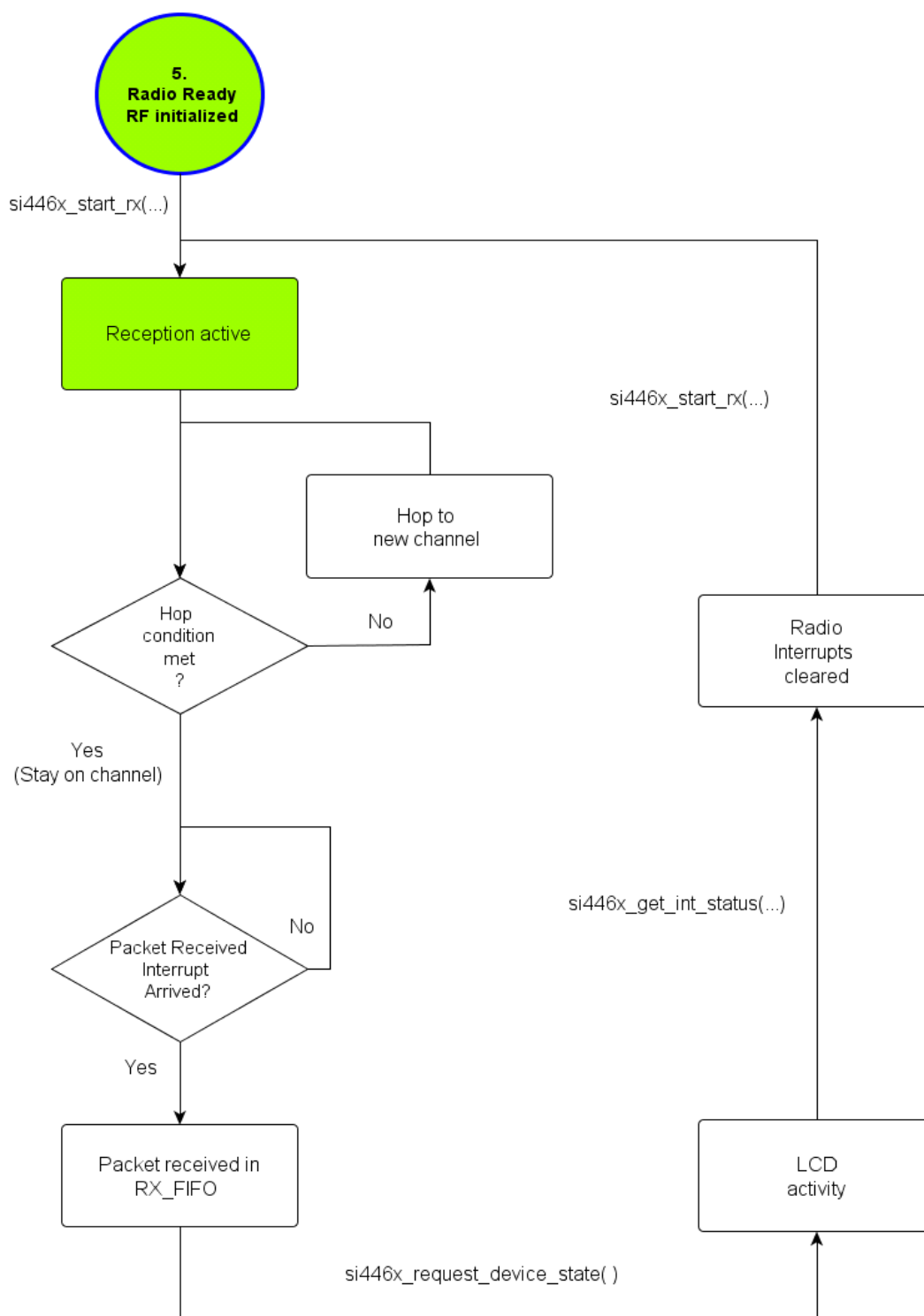


Figure 54. Packet Reception with Automatic RX Hopping Flowchart

10.14. Packet Reception with Manual Hopping Capability

The purpose of the standard packet reception with manual hopping feature example code is to demonstrate how the radio can receive packets in FIFO mode and how rapidly it scans frequency channels to search for signals. Once the device is configured into the RX state, it automatically starts hopping through the pre-configured channels on different frequencies. The RX_HOP API command provides the fastest method for hopping from one channel to another channel but it requires more management by the host MCU. Using the RX_HOP command, the turn-around time is 75 μ s. The timing is faster with this method than using either the START_RX or RX_HOP_CONTROL (automatic hopping) API commands because one of the calculations required for the synthesizer calibrations is offloaded from the radio chip. The arguments of the RX_HOP command are calculated by the Wireless Development Suite. They must be stored by the host that provides them for the radio.

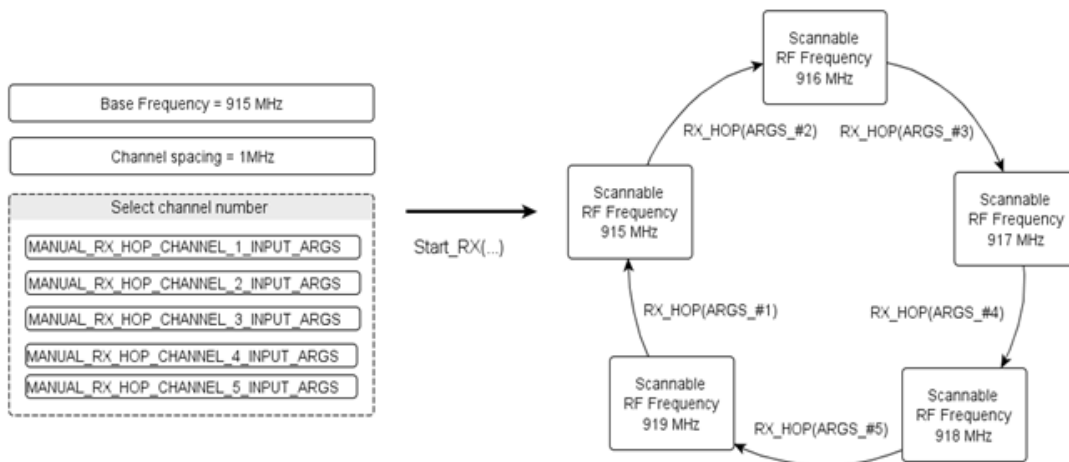


Figure 55. General Working Mechanism of the Manual RX Hopping

Since the host MCU has to provide the pre-calculated synthesizer arguments for the radio, such as the integer divide and fractional number of the fractional-N PLL, the host MCU needs to handle manually how much time the radio will spend in channels searching for signals. In every channel, the radio waits for the sync word detected successfully. If the timer supervised by host MCU expires without detecting the synch word then the host MCU will control the radio to hop to the next frequency channel manually. Once the correct bytes of sync word are received, the receiver stays on the channel in order to receive the payload of the packet and to fill it into the RX FIFO. The radio will not make a decision on staying on a channel or hopping to the next one automatically. There are not any configurable hop conditions for that purpose. The host MCU is responsible for determining whether to continue hopping or to stay on a particular channel. The MANUAL_RX_HOP_CHANNEL_X_INPUT_ARGS directive imported to the application in the "radio_config.h" header file as project specific setting contains the input arguments for the RX_HOP API command for one dedicated frequency channel. It determines the values of `FREQ_CONTROL_INTE`, `FREQ_CONTROL_FRAC2`, `FREQ_CONTROL_FRAC1`, `FREQ_CONTROL_FRAC` API properties and `VCO_CNT1` and `VCO_CNT0` properties.

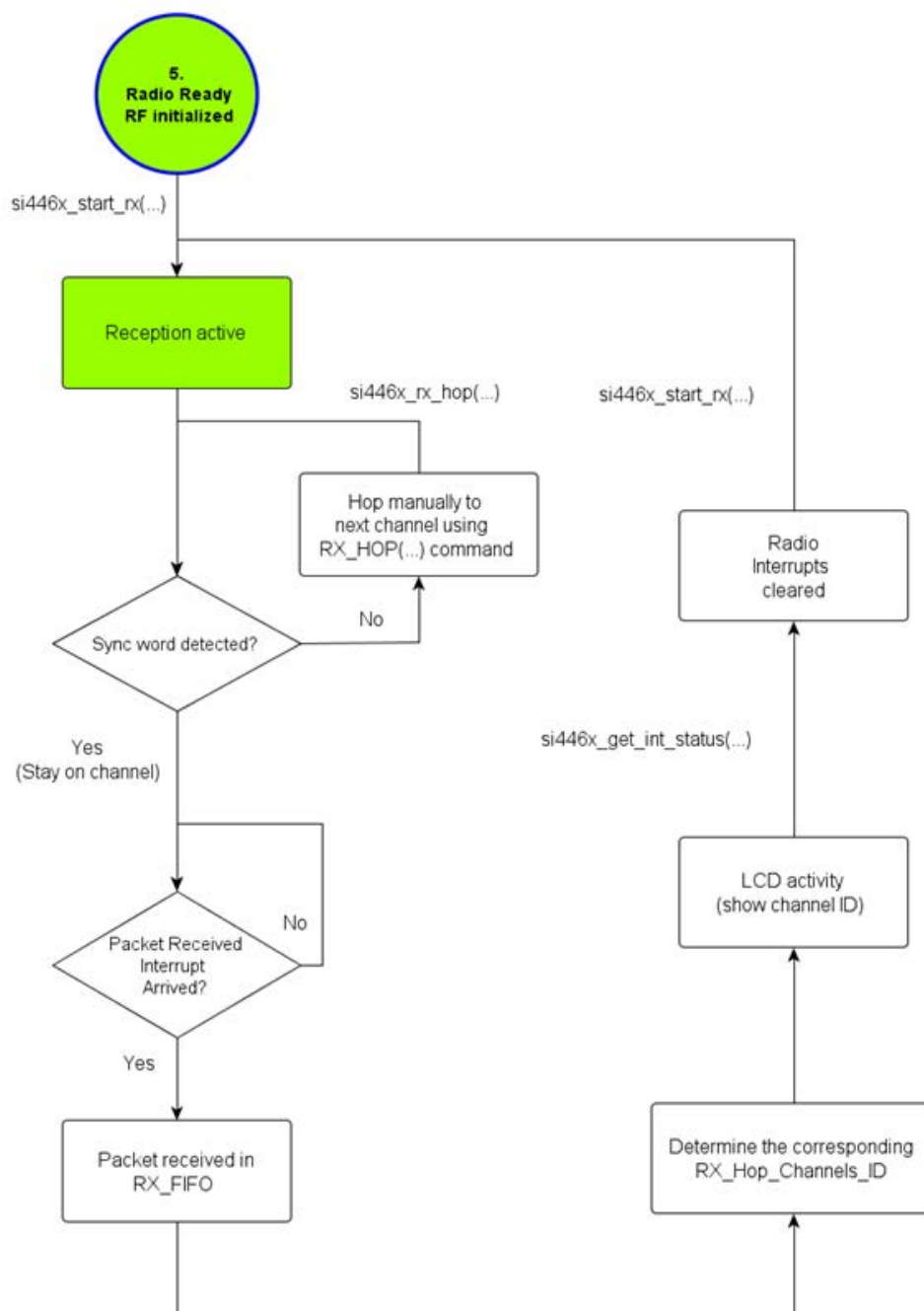


Figure 56. Packet Reception with Manual Rx Hopping

10.15. Continuous Transmission of Custom Amount of Standard Packets

The purpose of the standard packet transmission example code is to demonstrate how the radio can send packets in FIFO mode continuously. If the first button is pressed on the Wireless Motherboard then the host MCU will load the pre-defined content, namely "BUTTON1" in TX_FIFO and after that will send it. Pressing the button once prompts the radio to send the specified number of the same packets sequentially.

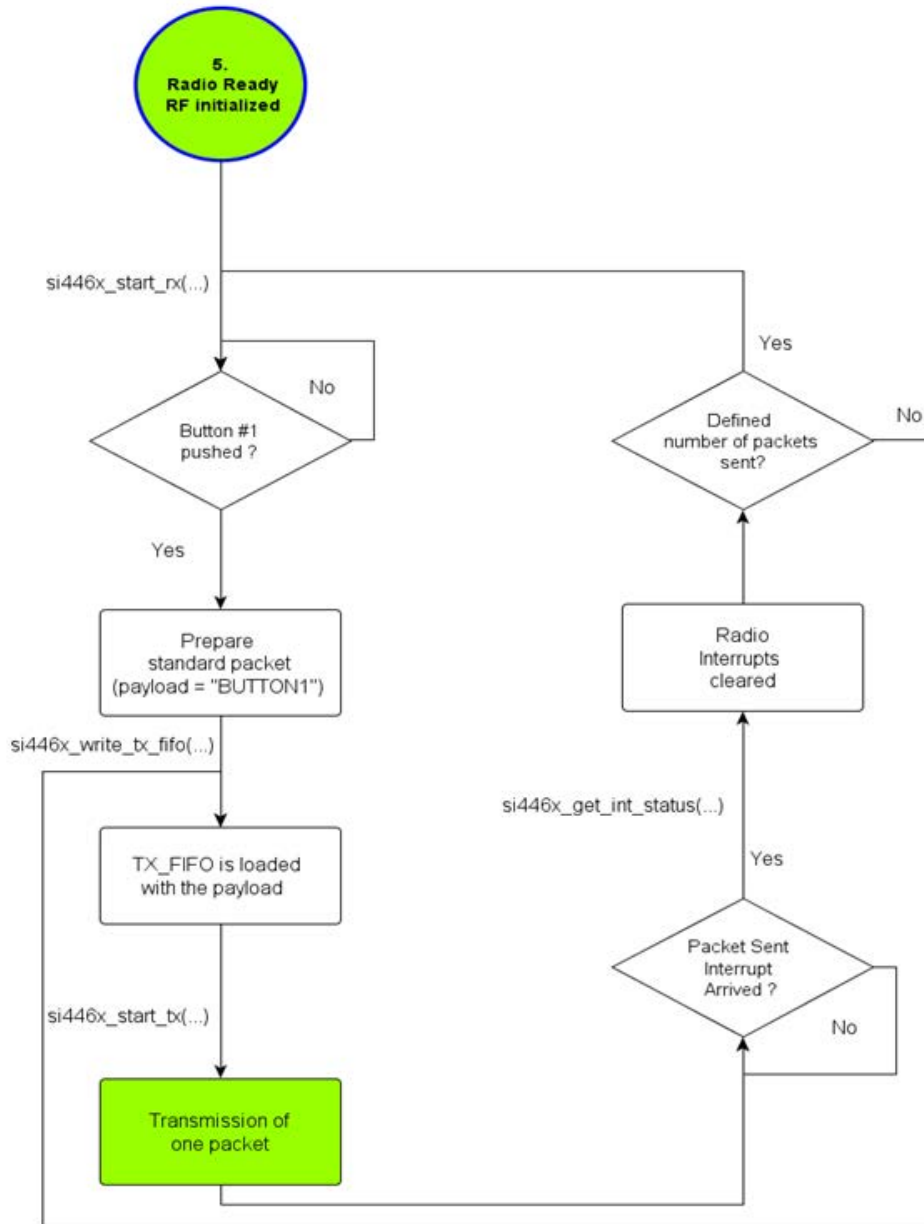


Figure 57. Continuous Transmission Flowchart

This project is the transmitter side of the low duty cycle receiver project. It can send a custom amount of packets in order to satisfy the needs of the receiver side namely to determine the minimum number of packets to be transmitted so that the receiver working in low duty cycle mode can certainly receive the packet. In the LDC mode the radio sleeps a certain amount of time called "Sleep time" then wakes up and listens for the signal in the "RX time".

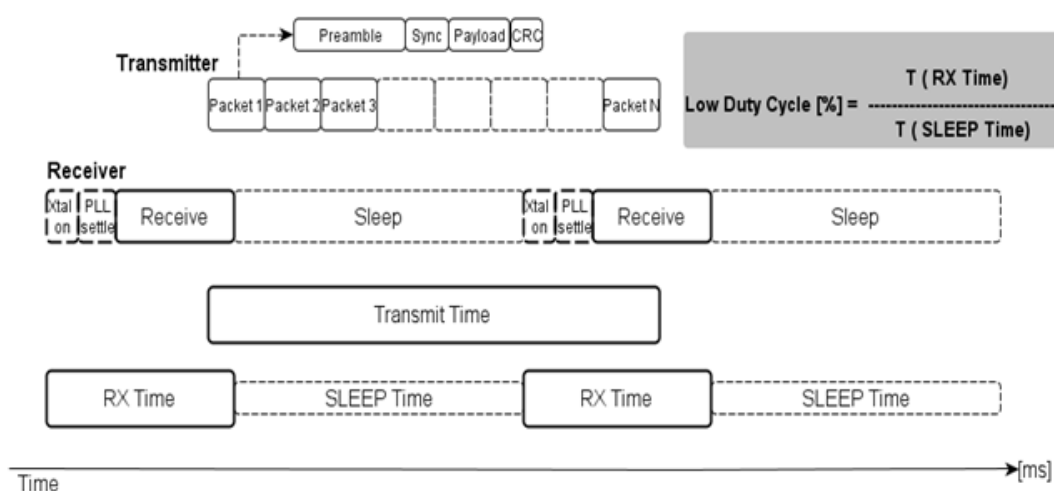


Figure 58. General Usage of the Continuous Transmission

In order to calculate the minimum number of packets to be transmitted, it is necessary to know how the different time periods, such as the “Transmit Time”, the “RX Time” and “Sleep Time” relate to one another in the worst case scenario. If the transmitter starts to transmit just after the receiver entered sleep mode, the transmitter needs to transmit while the receiver is in sleep mode plus the receiver wakes up and still a packet needs to be transmitted.

$$\min(\text{Transmit Time}) = T(\text{SLEEP Time}) + T(\text{XTal on} + \text{PLL settle}) + T(\text{one packet})$$

10.16. Standard Packet Reception with Low Duty Cycle Capability

The purpose of the standard packet reception example code using the low duty cycle mode is to demonstrate how the radio can receive packets in FIFO mode when the radio chip is continuously switching between the RX state and the SLEEP state. The receiver periodically wakes itself up to work on RX state. If a valid preamble is not detected or an entire packet is not received, the receiver returns to the Sleep state and remains in that mode until the beginning of the next RX state. If a valid preamble or sync word is detected, the receiver receives the entire packet. Wireless Development Suite makes it possible to configure the length of the “RX time” and the “SLEEP time”. In order to calculate the minimum length of the “RX time”, it is necessary to know how the different time periods, such as the “Transmit Time”, the “RX Time” and “Sleep Time”, relate to one another in the worst case scenario. If the receiver just settled right after the preamble is transmitted by the transmitter, the receiver couldn’t get this packet. Consequently, the receiver has to stay awake as long as this packet is transmitted, wait for the delay between the transmitted packets plus for the preamble is transmitted from the next packet to trigger the preamble detection circuit.

$$\min(\text{RX Time}) = T(\text{packet w/o preamble}) + T(\text{packet delay}) + T(\text{preamble threshold})$$

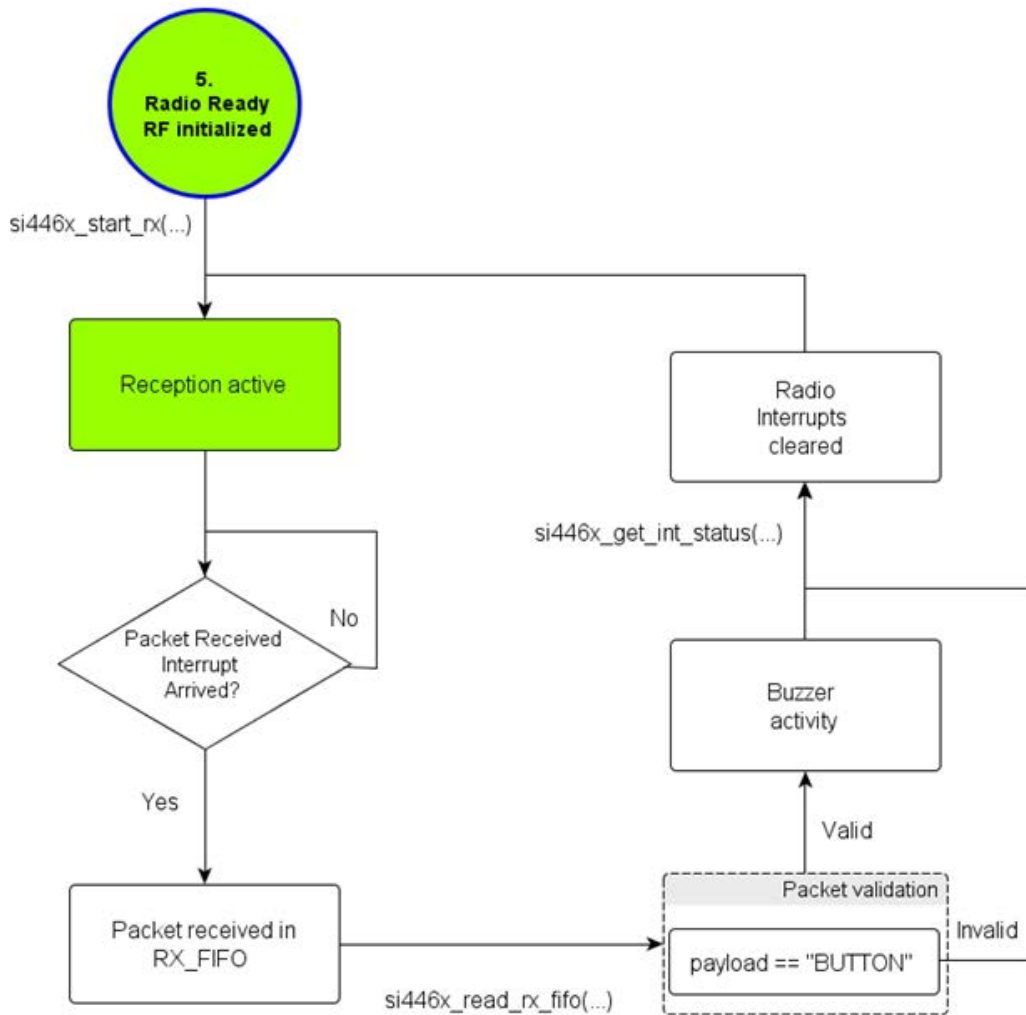


Figure 59. Standards Packet Reception with Low Duty Cycle Flowchart

10.17. Long Packet Transmission

Applications requiring packet length greater than the TX/RX FIFO sizes (64 bytes) may use the long packet feature of the radio. In such a case, TX FIFO Almost Empty interrupt should be monitored for proper timing to fill the TX FIFO. To determine when the Almost Empty interrupt should actually occur, a threshold level can be set. As for the TX side of the link, the TX FIFO Almost Empty and Packet Sent interrupts has to be enabled during initialization. Upon a button push, the first 64 bytes are filled into the TX FIFO and the host MCU starts waiting for a TX FIFO Almost Empty interrupt. When the interrupt arrives, the host MCU starts and fills TX_THRESHOLD number of bytes into the FIFO, and then goes back to the state in which it is waiting for the next TX FIFO Almost Empty IT, and so on. If the remaining bytes are less than the TX_THRESHOLD, they are put into the FIFO and the host MCU waits for the packet sent interrupts.

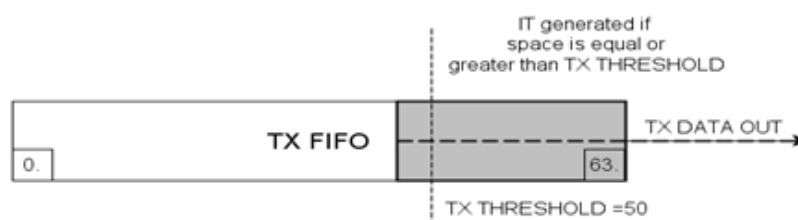


Figure 60. Occurrence of the TX FIFO Almost Empty Interrupt

10.18. Long Packet Reception

Applications requiring packet length greater than the RX FIFO sizes (64 bytes) may use the long packet feature of the radio. In such a case, RX FIFO Almost Full interrupts should be monitored for proper timing to read the RX FIFO. To determine when the Almost Full interrupt should actually occur, a threshold level can be set. As for the RX side of the link, the RX FIFO Almost Full and Packet Sent interrupts have to be enabled during initialization. After sending a START_RX command, the host MCU begins waiting for the RX FIFO Almost Full IT. When it arrives, it reads out RX_THRESHOLD number of bytes from the RX FIFO and continues waiting for the next RX FIFO Almost Full interrupt, etc. If the expected bytes are less than RX_THRESHOLD, the host MCU should wait for the packet received interrupt.



Figure 61. Occurrence of the RX FIFO Almost Full Interrupt

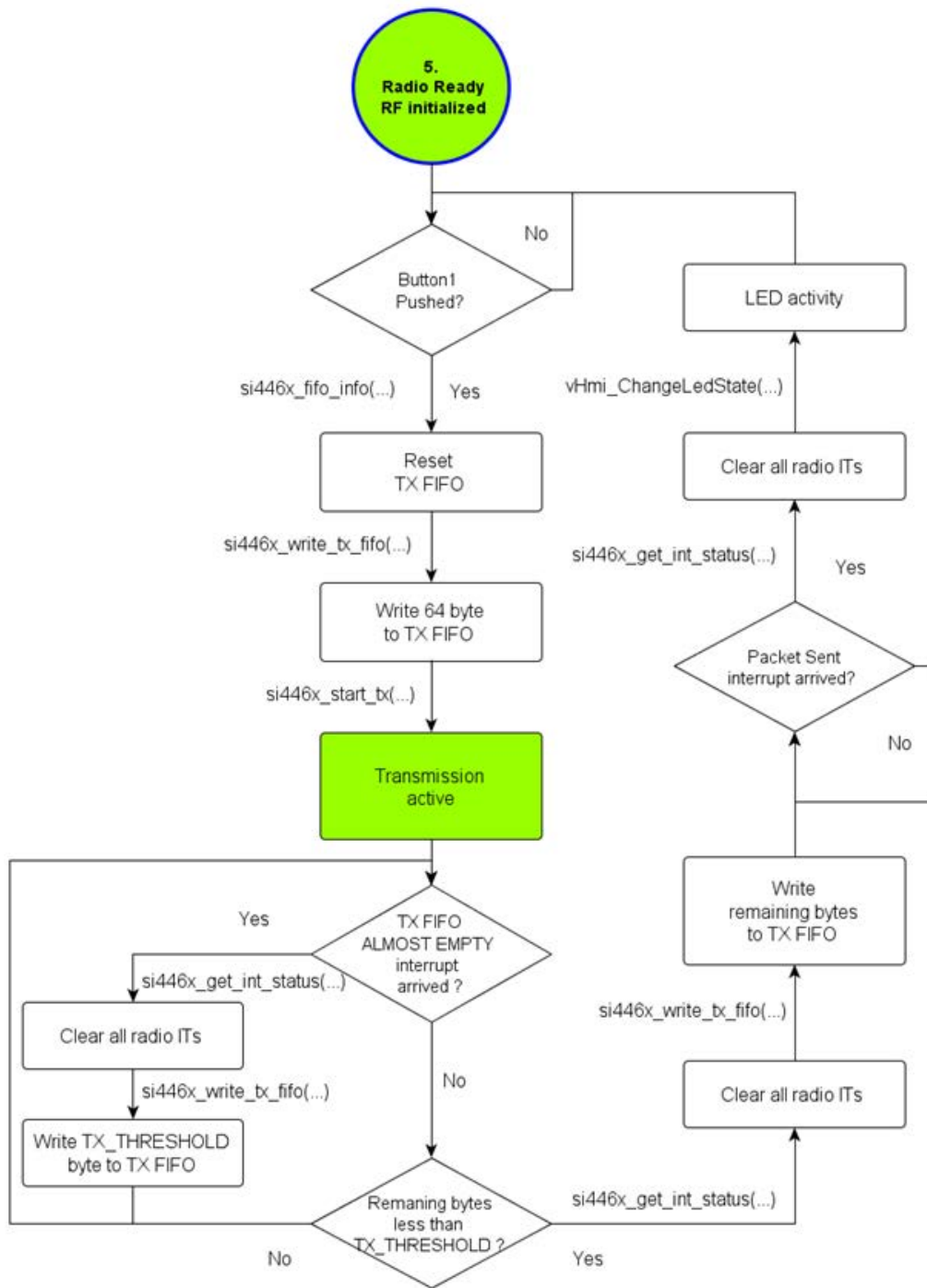


Figure 62. Long Packet Transmission Workflow

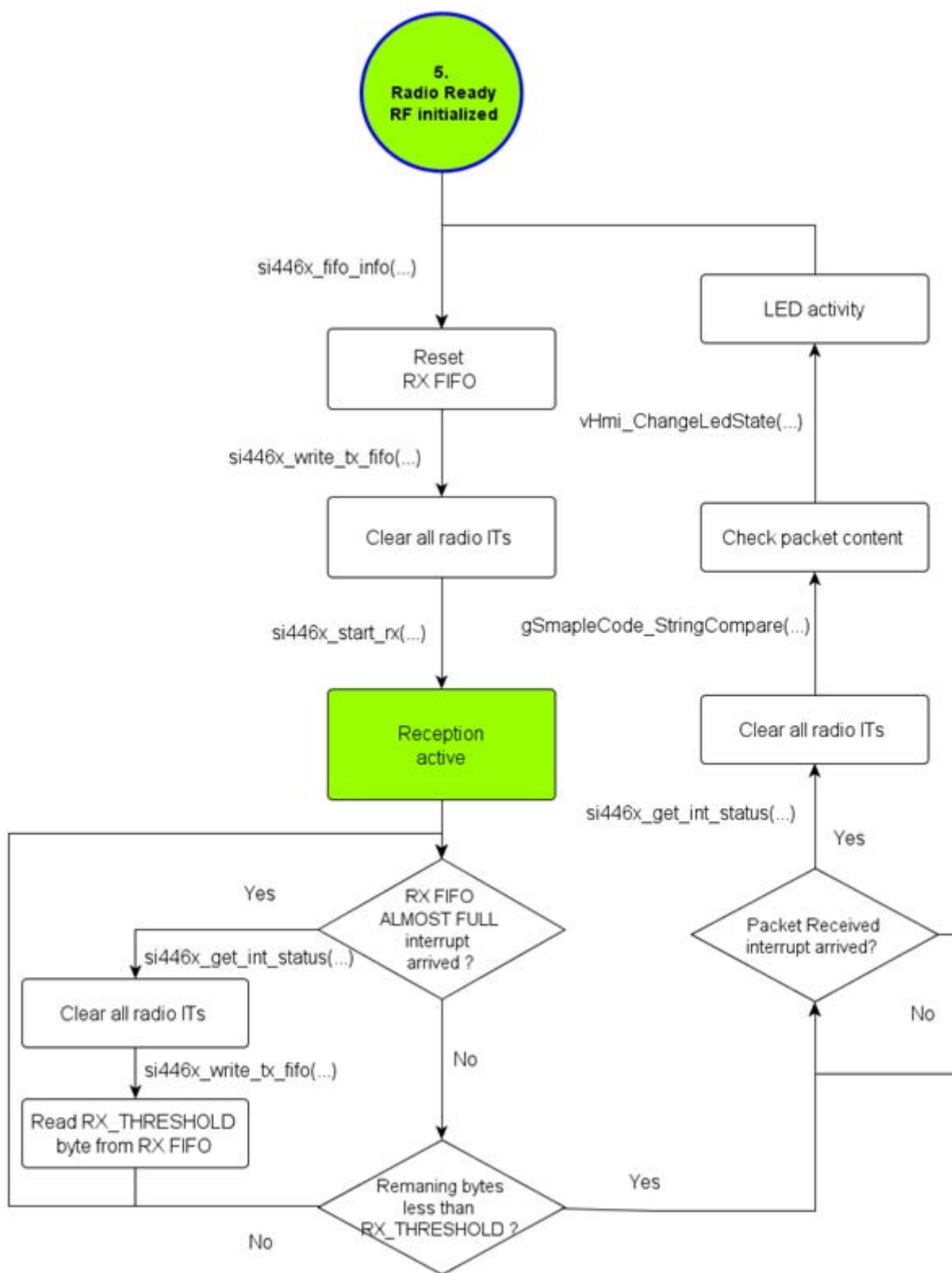


Figure 63. Long Packet RX Flowchart

10.19. Cooperation Between the Example Projects

Since several example projects based on packet-related communication are introduced in the Wireless Development Suite, it is essential to know which projects can communicate with each other in order to create a working one way-link. They are customizable in RF perspective such as by modulation type, data rate, deviation, etc. The payload is also customizable due to 5 configurable fields provided by the packet handler. AN632 can provide information about the projects' behavior and their purpose.

Table 18. Projects' Cooperation Description

Projects' Cooperative Activity						
	Standard Packet TX	Standard Packet RX	Custom Packet TX	Custom Packet RX	Packet Match RX	Frequency Hop RX
Standard Packet TX	N/A	✓	X	✓	✓	✓
Standard Packet RX	✓	N/A	✓	X	X	X
Custom Packet TX	X	✓	N/A	✓	✓	✓
Custom Packet RX	✓	X	✓	N/A	X	X
Packet Match RX	✓	X	✓	X	N/A	X
Frequency Hop RX	✓	X	✓	X	X	N/A

Table 19. Projects' Cooperation Description

Projects' Cooperative Activity						
	Standard Packet TX	Manual Hop RX	LDC TX	LCD RX	Long TX	Long RX
Standard Packet TX	N/A	✓	X	X	X	X
Manual Hop RX	✓	N/A	X	X	X	X
LDC TX	X	X	N/A	✓	X	X
LCD RX	X	X	✓	N/A	X	X
Long TX	X	X	X	X	X	✓
Long RX	X	X	X	X	✓	X

10.20. Empty Project

The empty project is created to help users write custom firmware. The project follows the convention for directory structure introduced in the sample projects. It contains driver modules for the radio and MCU peripherals as well as a default MCU initialization procedure. The porting of an example project to an MCU of choice can be done easily thanks to the layered approach of the project structure. This reduces the effort required to compile the code for other architecture, as only the low-level functions must be modified. The general structure of the project can be seen in the following figure. All the tasks are separated into two groups: the Hardware Initialization part and the Main Process part. Host MCU-related tasks initialize the physical interface between the radio and the controller unit, including the SPI lines (SCLK, SDI, SDO, NSEL), general I/O ports (SDN, NIRQ). The radio-related tasks prepare the radio for the communication and put the radio in ready state.

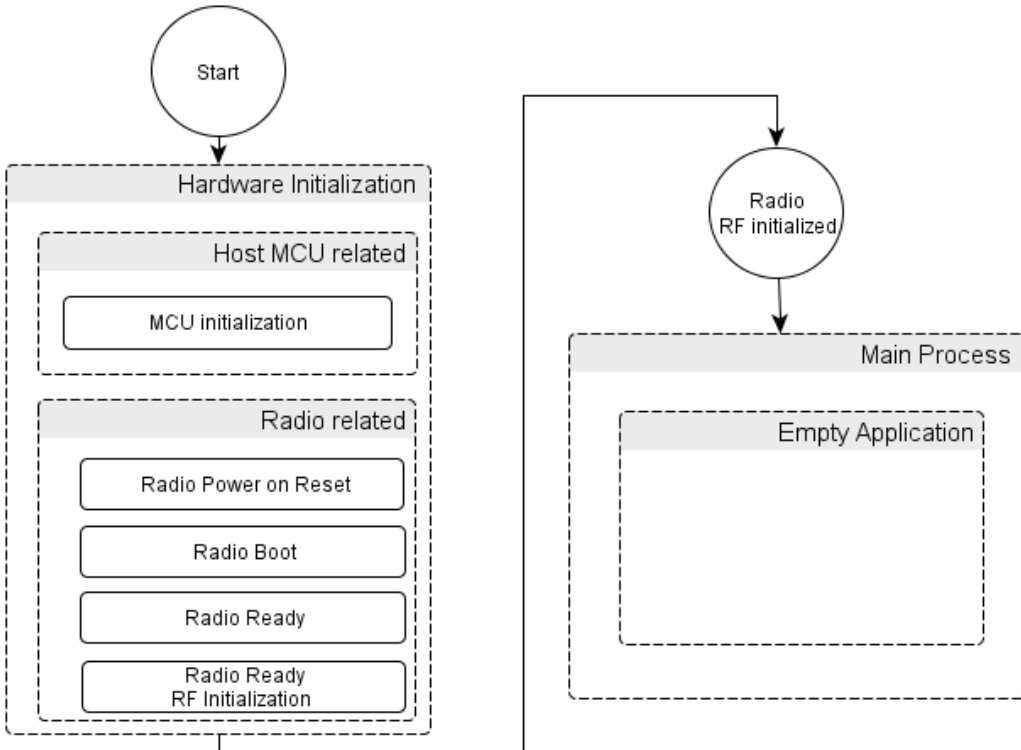


Figure 64. Structure of the Empty Project

The porting of an example project to an MCU of choice can be done easily thanks to the layered approach of the project structure. This reduces the effort required to compile the code for other architecture, as only the low-level functions must be modified.

The following drivers will be modified:

- `si_toolchain.h`, `hardware_defs.h`, `platform_defs.h`, `application_defs.h`
These header files contain definitions for the 8051 architecture and the Silicon Labs hardware platform. They may be modified according to the new architecture and hardware.
- `spi.c`, `spi.h`
The SPI driver module will be supplied to enable the communication with the radio.
- `radio_hal.c`, `radio_hal.h`
The radio hardware abstraction layer may be adjusted, as the GPIOs, NIRQ and SDN pins are defined in this file.

The above mentioned files may not cover all requirements for porting the project to other MCU, as it depends on what is to be ported and which other drivers are used by the project. The compiler tool chain setup, the appropriate startup codes, and linker scripts are out of the scope of this section; the user is responsible for providing them as appropriate for the given architecture.

11. Additional Resources

- AN104: Integrating Keil 8051 Tools into Silicon Labs IDE
- AN796: Wireless Development Suite General Description
- AN632: WDS User's Guide for EZRadioPRO® Devices
- Si406x Data Sheet
- Si4362 Data Sheet
- Si4464/63/61/60 Data Sheet
- Si4438 Data Sheet

Silicon Labs

Simplicity Studio™4



Simplicity Studio

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



IoT Portfolio
www.silabs.com/IoT



SW/HW
www.silabs.com/simplicity



Quality
www.silabs.com/quality



Support and Community
community.silabs.com

Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required, or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, ClockBuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, ISOModem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. Wi-Fi is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.



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

<http://www.silabs.com>