μSDR User's Manual
Version 0.1

Ye-sheng Kuo

Sep 2012

Contents

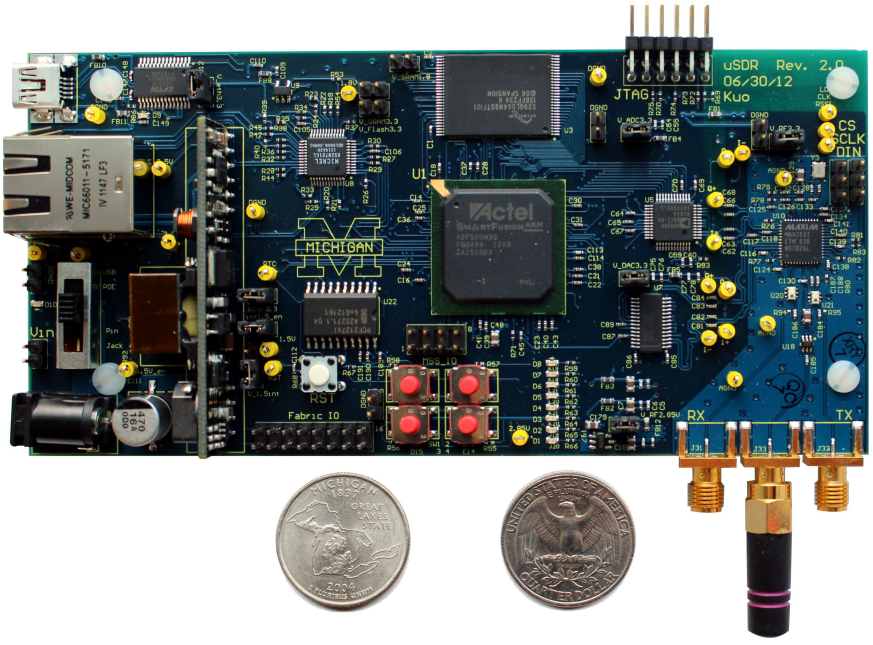
Chapter 1

Overview

μSDR is a software-defined radio based on Microsemi's SmartFusion. Unlike common FPGAs, the SmartFusion is a device which incorporates a hard core ARM Cortex M3 and a flash-based FPGA. Flash-based FPGA technology offers low static power, no reconfiguration in boot stage and memory retention when in power down mode. These benefits make it a better candidate for low-power wireless research. We present the μSDR, a true battery-powered SDR platform which is suitable for portable and deployable hand held device research.

1.1 Features

1. 2.4 - 2.5 GHz ISM band
2. Dual channel 80 MS/s, 8-bits ADC
3. Dual channel 40 MS/s, 8-bits DAC
4. 16 MB External PSRAM
5. 8 MB External Flash
6. ±3 ppm TXCO
7. Ethernet, USB interfaces
8. 24 User defined IO, 8 LEDs and 4 switches
9. Battery-powered



Chapter 2

Hardware

NOTE: needs a top level diagram. That would make the connections a lot easier to follow.
IDEA: combine all of the IO connection tables into one super table so all wiring is in one spot.
The major components of μSDR are the following:

Microsemi Smartfusion
The procssing core of μSDR. It incorporates an analog computing engine, a flash-based FPGA and a hardcore ARM Cortex-M3.

MAXIM MAX2831
The MAXIM 2831 is a 2.4 GHz ISM band RF frontend. This chip integrates common RF blocks (power amplifier, filters, voltage control oscillator, etc.) into a single chip.

Analog Device AD9288
8-bits Dual channel high speed ADC. Three variants are available. The one which is currently employed on μSDR is 80 MSP/s.

MAXIM MAX5189
8-bits Dual channel 40 MHz DAC. Both channels share the same input but latching the data using different edges of clock input.

Micrel KS8721CL
100Base-TX Physical Layer Transceiver providing RMII interfaces to MACs and switches.

FTDI FT232R
USB to UART interface. It connects to the UART0 of SmartFusion device.

External Mem
μSDR has 2 external memories which are 16 MB PSRAM and 8 MB flash installed.

2.1 Power

2.1.1 Power Rails

μSDR uses many different power rails for many purposes. Every power rail is produced by an LDO (Low Drop-Out regulator).

1. 3.3 V, the major power supply for **Smartfusion, ADC, DAC, RF, Ethernet, USB, RTC, PSRAM, and FLASH**
2. 2.85 V, **RF** only.
3. 1.8 V, **PSRAM** only.
4. 1.5 V, **Smartfusion(Core)** only.

2.1.2 Power Source

μSDR can be powered by USB, PoE, exposed headers or power jack. A 4-way switch is used to select the input power source.

2.1.3 1.5V on-chip regulator

The SmartFusion integrates a 1.5 V on-chip regulator. This regulator could be turned on by triggering a specific IO or software interrupt. Similarly, this regulator could be turned off by setting a specific register.

Jumper P15 ("int1.5_en") must be jumpered in order to enable the on-chip 1.5 V regulator. Jumper P16 ("V_1.5int") is the 1.5 V power selector. Jumpering the upper 2 pins powers the μSDR using the external regulator. In contrast, jumpering the bottom 2 pins causes μSDR to use the on-chip 1.5 V regulator.

2.1.4 Enabling/Disabling Power

μSDR offers many jumpers to separate the power domain across subsystems. User can manually disconnect the power supplies by removing the jumpers. These jumpers also provide convenient locations to measure the current through a specific chip.

Note: 1.5 V and 3.3 V on SmartFusion and Ethernet are required for the μSDR to operate properly.

NOTE: this needs to be further explained with a table of what jumpers power what.

2.2 RF Frontend

The MAX2831 provides a 3-wire SPI interface as well as a 7-bit wide gain control bus. Every register can be written through the SPI interface, however, the SPI interface doesn't allow for reading the current registers' value. TX baseband gain, RX baseband gain and RX LNA gain can be controlled using these registers and 7-bit bus. Since the SPI interface clocks a single bit at a time, it has longer latency when setting the gain control than the 7-bit bus. Therefore, AGC benefits from the parallel bus. In order to enable the parallel bus gain control, registers 8 D12 and 9 D10 must be asserted. In the receiving mode, bits B7 and B6 control the LNA gain setting and bits B5˜B1 control the baseband VGA gain.

Two additional control wires (RX\TX, SHDN) control the modes. Available modes are listed in Table 31 in datasheet [3].

The analog RSSI signal is connected to a low-power serial ADC [9] that operates up to 1 MSP/s. The maximum output of serial ADC is 223 (3.3 V) instead of 255. Therefore, the step size of serial ADC is **14.8 mV**.

The MAX2831 separates the TX/RX path. Since it cannot fully duplex communication, an RF switch [8] is used to allow a single antenna. If desired, users can use separate antennas by applying the three small changes listed below.

Instructions for Using Separate Antennas:

1. Remove C184, C186.
2. Short R94, R95.
3. Install antennas on J31, J33.

2.2.1 Test Points

Various test points are available on μSDR. Analog signals include TX/RX I/Q channels, LD, CLK and RSSI. P12, P13 (on the edge of μSDR) provide digital test points for SPI interface of MAX2831 and ADC081S101.

NOTE: need a better table for this. It's not clear which pins exactly correspond to which SPI signals.

2.2.2 IO connection

MAX2831 interface

Name	Physical Connection	Direction
RX\TX	L18	Out
SHDN	M18	Out
$\bar{C}S$	H20	Out
DATA	J22	In
CLK	L22	Out
B1 (VGA, LSB)	D7	Out
B2 (VGA)	E8	Out
B3 (VGA)	C4	Out
B4 (VGA)	C5	Out
B5 (VGA, MSB)	D8	Out
B6 (LNA, LSB)	C7	Out
B7 (LNA, MSB)	C8	Out

RSSI Serial ADC (ADC081S101) interface (SPI)

Name	**Physical Connection**	**Direction**
$\bar{\text{C}}$S	H17	Out
DATA	H22	In
CLK	H18	Out

2.3 ADC

Channels A and B of the AD9288 [1] are connected to the In-phase and Quad-phase channels of the MAX2831, respectively. The input common-mode voltage of the AD9288 is 0.3∗VDD which is 0.99 V. However, the common-mode voltage of the MAX2831 is not able to be tuned to 0.99 V. A voltage reference chip, the MAX6061 [5], and a voltage divider are used to alter the common-mode voltage.

The output of the AD9288 on μSDR is pre-configured to 2's complement output. Therefore, $D7_{A,\ B}$ are sign bits for each channel. Deassert signal S1 to put the AD9288 into standby mode. More detail can be found in Table 4 in datasheet [1].

2.3.1 IO connection

Name	**Physical Connection**	**Direction**
$D0_A$ (I, LSB)	C21	In
$D1_A$	D21	In
$D2_A$	B20	In
$D3_A$	C19	In
$D4_A$	G19	In
$D5_A$	F19	In
$D6_A$ (I, MSB)	G21	In
$D7_A$ (I, Sign)	G20	In
$D0_B$ (Q, LSB)	K17	In
$D1_B$	J17	In
$D2_B$	F21	In
$D3_B$	F20	In
$D4_B$	G18	In
$D5_B$	G17	In
$D6_B$ (Q, MSB)	E18	In
$D7_B$ (Q, Sign)	F17	In
CLK	A17	Out
S1	D18	Out

2.4 DAC

Similarly, the MAX6061 voltage reference chip is deployed on the transmission path to pull the common-mode voltage. The MAX5189 [4] uses both edges of the clock to update both channel outputs. The maximum clock input frequency of the MAX5189 is 40 MHz. The combination of DACEN and PD determines the operation mode, more detailed information can be found in Table 1 of the datasheet [4].

2.4.1 IO connection

Name	Physical Connection	Direction
D0 (LSB)	E1	Out
D1	F3	Out
D2	G4	Out
D3	H5	Out
D4	H6	Out
D5	J6	Out
D6	B22	Out
D7 (MSB)	C22	Out
CLK	A18	Out
DACEN	F1	Out
PD	G2	Out

2.5 User's IO

μSDR has eight user-defined LEDs and four push-button switches. The LEDs and switches are **active low**. The LEDs/switches are connected to the Fabric, however, users can route the connection to the MSS IO. In addition, two IO banks also available on μSDR. A 16-pin wide interface is connected to the Fabric, whereas an 8-pin bus is connected to fixed[1] MSS GPIOs. Libero SoC must be configured to enable the MSS GPIO.

2.5.1 IO connection

Name	Physical Connection	Direction
LED0	J20	Out
LED1	J19	Out
LED2	K20	Out
LED3	K21	Out
LED4	L20	Out
LED5	L21	Out
LED6	K18	Out
LED7	K19	Out
SW1	J21	In
SW2	B19	In
SW3	D15	In
SW4	E14	In

[1] GPIOs on Fabric are able to route to MSS IO Pads, however, GPIOs on MSS cannot be routed to Fabric.

Name	Physical Connection	Direction
Fabric IO_0	N6	In/Out
Fabric IO_1	M6	In/Out
Fabric IO_2	P1	In/Out
Fabric IO_3	P2	In/Out
Fabric IO_4	N3	In/Out
Fabric IO_5	N2	In/Out
Fabric IO_6	M2	In/Out
Fabric IO_7	M1	In/Out
Fabric IO_8	M4	In/Out
Fabric IO_9	L3	In/Out
Fabric IO_{10}	L2	In/Out
Fabric IO_{11}	L1	In/Out
Fabric IO_{12}	L5	In/Out
Fabric IO_{13}	K4	In/Out
Fabric IO_{14}	L6	In/Out
Fabric IO_{15}	K6	In/Out
MSS IO_0	T3	In/Out
MSS IO_1	V3	In/Out
MSS IO_2	U3	In/Out
MSS IO_3	T4	In/Out
MSS IO_4	AA2	In/Out
MSS IO_5	AB2	In/Out
MSS IO_6	AB3	In/Out
MSS IO_7	Y3	In/Out

2.6 TCXO

The PCF2127A [6] is a temperature-compensated low-frequency oscillator. It provides ±3ppm stability over a wide range of temperature and it replaces the 32.768 KHz crystal oscillator replaces what crystal oscillator?. The configuration interface is connected to the MSS SPI bus 1[2] and the interrupt is connected to a Fabric IO. A external coin-cell battery provides an additional power source for the PCF2127A. μSDR automatically switches the power source if V_{DD} is less than a certain threshold. Less than what threshold?

2.6.1 IO connection

Name	Physical Connection	Direction
INT	B7	In
D_{IN}	T17	Out
D_{OUT}	V19	In
CLK	AA22	Out
$\bar{CS}$	W21	Out

[2]It's a fixed connection and needs to be initiated in Libero SoC MSS configuration tool.

2.7 Ethernet

2.7.1 Power over Ethernet

μSDR can be powered by Ethernet. The PoE module on the μSDR is the AG9205S [7] and it provides up to 13 W power output. The default power setting on μSDR is configured to 3.84 W, which is equivalent to 768 mA maximum current. The setting can be changed by replacing the configuration resistor R50. More power ratings are available in Table 2 in datasheet [7].

2.7.2 Disabling KSZ8721

The Ethernet transceiver KSZ8721 on μSDR is enabled by default. The transceiver draws about 50 mA when in an idle state. In low-power applications when the transceiver is not needed, it can be turned off[3] by patching the μSDR.

Instructions to Disable the KSZ8721 Ethernet Module:

1. Remove R36.
2. Put 10 K resistor on R33.

2.7.3 IO connection

Name	Physical Connection	Direction
CLK (50 MHz)	E3	In

[3] The Ethernet subsystem cannot be disabled by removing the jumper P7 ("V_eth3.3"). Doing this results in the SmartFusion working improperly.

Chapter 3

802.15.4 Application

3.1 Fabric Architecture

802.15.4 application integrates the basic transceiver functionality into Fabric. User can initiate transmission or receiving by simply writing to specific registers. Figure 3.1 is a simplified model of state machine. The details for each state will be discussed in following.

3.1.1 Radio states

Off Mode

Once the μSDR exits global system reset, radio starts with **Off** state. In this state, the radio peripherals (RF frontend, ADCs and DAC) are in the lowest power state to save energy. Radio exits this state if the following event has been triggered.

1. **Deasserting Radio off/on register:** Radio exits **Off** mode and a series of hardware initializations take place at this stage. Once the discrete chips are ready, the radio arrives **Idle Listening** mode.

Idle Listening Mode

In this mode, radio has RF frontend on in RX mode and ADCs actively running. Radio keeps decoding wireless signal and trying to identify any incoming packet. In addition, radio is ready to initiate a packet transmission in this state. Radio exits this state if the following event has been triggered.

1. **Asserting Radio off/on register** Radio turns-off all peripherals and entering the **Off** mode to save energy.
2. **Incoming packet detected** Once a valid length field has been decoded, radio enters **Receiving** mode to ensure receiving a completed packet.
3. **Initiating packet transmission** The start transmission command from processor or Fabric itself trigger the radio into **Transmission** mode.

Receiving Mode

Since the radio is not transmitting in this state, DAC has been turned-off to save energy. In the mean time, RF frontend and ADCs actively running and receiver blocks decode wireless data. Receiver performs a basic frame filtering and storing the incoming bytes into fifos. Radio exits this state if the following event has been triggered.

1. **Receiving completed** A packet has been received completely and this packet doesn't request ACK nor Forward. Radio backs to **Idle Listening**.
2. **ACK/Foward completed** Radio backs to **Idle Listening** if ACK/Forward[1] packet has been transmitted completed.

Transmission Mode

In transmission mode, ADCs has been turned-off and the receiver is put into reset. Transmitter starts reading the data from TX fifo and up converting through RF frontend. Radio exits this state if the following event has been triggered.

1. **Transmission completed** The TX fifo is empty indicates the transmission has been completed. Radio switches back to RX and going to **Idle Listening**.

[1] 802.15.4 standard requires 192 μs between transmissions (ACK/Forward). The timing requirement is taken care in hardware.

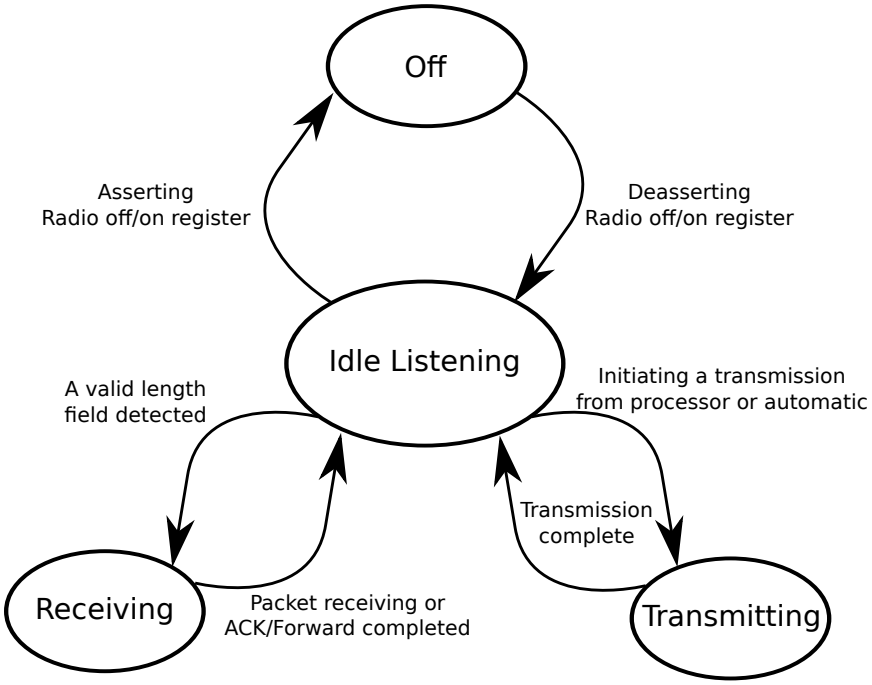


Figure 3.1: Simplified radio state diagram

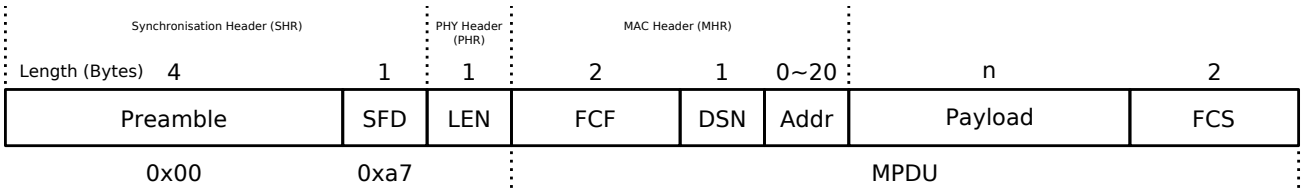


Figure 3.2: Frame format of a 802.15.4 packet

3.1.2 fifos

μSDR takes the advantage of high speed interconnect between processor and FPGA fabric. The processor loads/unloads the data from radio by accessing to specific addresses. The writing/reading operation is mapped to the fifo operation. With integrated PDMA, data operation can be even faster and requests less processor's resources. The following fifos can be found in 802.15.4 application.

1. **RX fifo**
2. **TX fifo**
3. **ACK fifo**
4. **FWD fifo**

RX fifo

The depth and width of RX fifo is 512 and 8-bits respectively, which is able to store 4 packets in maximum length. Unlike other fifos, RX fifo is an embedded fifo, whereas other fifos are synthesized "soft" fifos.

A typical 802.15.4 packet starts with 4 bytes of preamble (0x00), 1 byte of SFD (0xa7) and length field comes after SFD. Once a valid length field has been detected, receiver start storing the decoded byte into RX fifo until the packet ends. The last 2 bytes in the packet are FCS (Frame Check Sequence) field. These bytes are used to perform CRC (Cyclic Redundancy Check) to detect errors. However, FCS is not useful to processor. Since the CRC is embedded in fabric. Instead of storing the FCS field in RX fifo, receiver stors the averaged [2] RSSI value and CRC result.

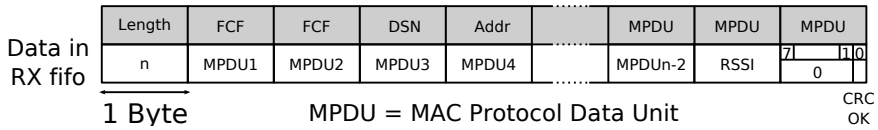


Figure 3.3: The contents of RX fifo

Figure 3.3 shows the contents stored in RX fifo. The last two bytes stored in the RX fifo are RSSI and CRC OK respectively. The RSSI value is unsigned 8 bytes value. As mentioned before, the step size of RSSI is 14.8 mV. (eg: 137 = 2.03 V in RSSI) CRC OK is a single bit in last byte. This bit is asserted if the packet passed CRC check and vice versa.

TX fifo

The depth and width of TX fifo is 128 and 8-bits respectively. In contrast to RX fifo, TX fifo contains only 1 packet every time. Transmitter loads preamble and SFD automatically, which occupy 5 bytes in depth. Thus, the truly available depth for user is $128 - 5 - 1$ (length) $= 122$ bytes. User only loads the "data" to TX fifo. The transmitter will automatically calculate corresponding FCS and appending it to the end of TX fifo.

ACK fifo

μSDR support hardware ACK (HACK). Parts of ACK frame is pre-stored in ACK fifo. Once the received packet has ACK request bit set, its DSN is passed to ACK fifo and fifo control starts calculating the FCS for ACK frame. User cannot access to ACK fifo directly.

FWD fifo

μSDR supports hardware packet forwarding. Receiver stores the decoded data into 2 different fifos - RX fifo and FWD fifo. However, the data stored in these fifos are slightly different in DSN field and FCS. μSDR uses DSN as a relay counter to prevent the forwarding packet last forever in the network. Only the packet has DSN which is less than threshold (0x7f) will be forwarded. Similar to TX fifo, FWD fifo is not able to be accessed by user.

[2] The value is averaged over 4 RSSI samples right before SFD has been detected. Approximately 5 μs.

3.1.3 Address Map

Address	Width	Bit Position	Direction	Registers
0x40050000	8	7:0	R	RX fifo
0x40050004	16	31:16	R	Src Pan ID[a]
0x40050004	16	15:0	R	Dest Pan ID[a]
0x4005000c	1	18	R	RX fifo empty
0x4005000c	1	17	R	ACK set[a]
0x4005000c	1	16	R	CRC correct[a]
0x4005000c	16	15:0	R	FCF[a]
0x40050010	32	31:0	R	Src address LSB[a]
0x40050014	32	31:0	R	Src address MSB[a]
0x40050018	32	31:0	R	Dest address LSB[a]
0x4005001c	32	31:0	R	Dest address MSB[a]
0x40060000	8	7:0	W	TX fifo
0x40070000	1	8	R/W	Auto TX en
0x40070000	4	7:4	R	Radio mode
0x40070000	1	3	R/W	ACK en
0x40070000	2	2:1	R/W	AGC mode
0x40070000	1	0	R/W	Radio off/on
0x40070020	8	7:0	R/W	LED
0x40070050	1	0	W	ACK flush
0x40080000	14	13:0	W	MAX2831 Register 0[b]
0x40080004	14	13:0	W	MAX2831 Register 1[b]
0x40080008	14	13:0	W	MAX2831 Register 2[b]
0x4008000c	14	13:0	W	MAX2831 Register 3[b]
0x40080010	14	13:0	W	MAX2831 Register 4[b]
0x40080014	14	13:0	W	MAX2831 Register 5[b]
0x40080018	14	13:0	W	MAX2831 Register 6[b]
0x4008001c	14	13:0	W	MAX2831 Register 7[b]
0x40080020	14	13:0	W	MAX2831 Register 8[b]
0x40080024	14	13:0	W	MAX2831 Register 9[b]
0x40080028	14	13:0	W	MAX2831 Register 10[b]
0x4008002c	14	13:0	W	MAX2831 Register 11[b]
0x40080030	14	13:0	W	MAX2831 Register 12[b]
0x40080034	14	13:0	W	MAX2831 Register 13[b]
0x40080038	14	13:0	W	MAX2831 Register 14[b]
0x4008003c	14	13:0	W	MAX2831 Register 15[b]

[a] Information is only valid for the last received packet.
[b] Refer to datasheet [3] for detail information.

3.2 Radio Operations

3.2.1 Receiving

Once the radio is on, it normally resides in **Idle Listening** mode. Once a packet arrives, various interrupts are generated by receiver. User is able to respond toward different events. Reading the received data is straightforward. User just need to keep reading the RX fifo address. If the reading is performed in packet basis, the first byte readed out by user is usually the packet length. Thus, user can initiate the PDMA transfer for the rest of the packet.

The receiver performs simple frame filtering. Thus, the address related fields are extracted and able to be read through AHB interface directly. However, user has to note that the information is only valid for the last received packet. Also, any transmission (including ACK/Forward) put the receiver into reset reset, which flushes the registers' values.

3.2.2 Transmitting

Loading the data into TX fifo, user simply write to the TX fifo address. It's not necessary to initiate the transmission after the TX fifo is completed filled. User can load the data into TX fifo while the radio is currently transmitting. In order to initiate the data transmission, user has to assert a GPIO which connects to the transmitter. However, the radio is able to start transmitting only if the radio in the **Idle Listening**. Therefore, user has to check current radio state before asserting the GPIO line. Two methods are available to check the current radio state.

1. Radio mode register (0x40070000)
2. GPIO inputs

By accessing the radio mode register, user is able to identify the current radio mode. If current radio mode is **0**, user is able to assert GPIO line. In addition, "ready_to_transmit" signal is connected to an MSS GPIO line, which is another indicator to initate the transmission.

User has to be aware that the exact timing for radio start transmitting is not the time when GPIO has been asserted. Since radio normally in the **Idle Listening** mode, it takes time for the RF frontend to switch its mode from RX to TX. In μSDR, the time between GPIO asserts and actually transmitted is 2 μs.

If precise timing is required, μSDR offers a fixed timing (192 μs) auto transmission. If the **"Auto TX enable"** is asserted, radio starts transmitting after 192 μs of a packet is received. Automatic transmission is useful in concurrent transmission scenario.

3.2.3 ACK

μSDR supports hardware ACK. By default, μSDR has Auto ACK enabled. User can disable the Auto ACK by deasserting the ACK en register. The ACK frame has to be sent right after 192 μs if **all** the following conditions are met.

1. ACK request field is set
2. Packet passes CRC
3. Destination address match radio's address

μSDR is designed to support multiple address recognition. However, the address field in 802.15.4 standard could be 8 bytes long. Giving the space constraint, the address recognition is moved to processor. Therefore, the processor must complete the 3rd condition within 192 μs. If a packet meets the first two conditions, fabric starts preparing the ACK frame and initiating a down counter. If destination address doesn't match to radio's address, processor has to "flush" the outgoing ACK. "Flush" can be done by asserting the ACK flush register (0x40070050).

3.2.4 Forwarding

μSDR supports hardware forwarding. The forwarding is entirely built in fabric. Thus no processor is required. The packet is been forwarded if **all** the following conditions are met.

1. Destination address is equal to 0xfffe or 0xffff_ffff_ffff_fffe (depends on destination address mode)
2. DSN is less than 127
3. Packet passes CRC

The fabric increments its DSN and updating its FCS for each forwarding packet. The packet will be forwarded after 192 μs

3.2.5 Automatic Gain Control (AGC)

MAX2831 RF frontend doesn't provide AGC in package. However, user can set fixed gain through SPI interface or enabling the AGC in fabric. The detail of RX gain setting can be found in MAX2831 datasheet Table 26 [3]. To enable the AGC in Fabric, user has to write to AGC mode register. Two different AGC mode is available.

AGC mode register	AGC Mode	Description
0	Off	AGC is disabled
1	SFD-Latched	Gain setting is latched by the SFD detection
2	Reserved	Reserved
3	Contiuous	Gain setting keeps adapting to signal strength

3.3 Interrupts and Pin Activity

3.3.1 Receiving

Several critical pins are connected to MSS GPIO, which allows user to monitor and responding to radio events. In the receiving mode, SFD pin goes high after radio successfully decoded SFD field in the packet. Similarly, Length pin goes high after **valid length** [3] field has been decoded. Once the length field is corrupted, radio back to wait preamble and the SFD pin deasserts. Figure 3.4 shows a "complete" interrupt is added to the μSDR. The "complete" interrupt is a pulse signal, which asserts after a packet reception has been complete and deassert after 62.5 μs. Since the SmartFusion has PDMA integrated,

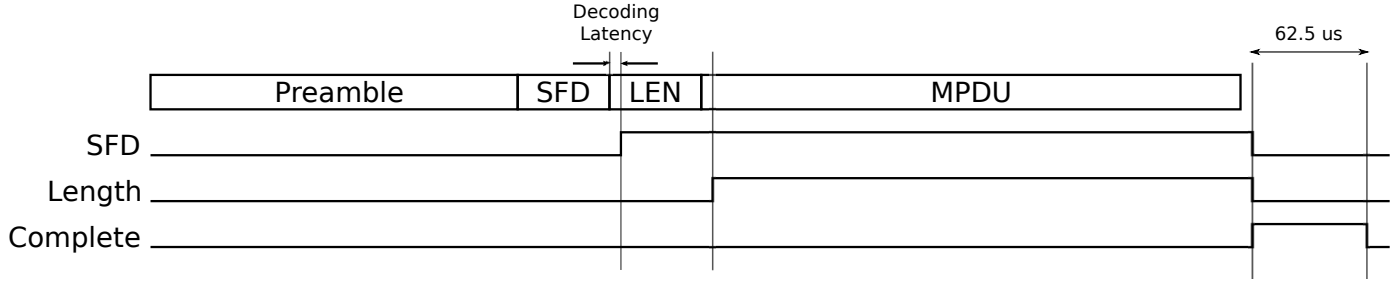


Figure 3.4: Pins activity during receiving

user can configure the complete as a interrupt to processor. Once the complete interrupt arrived, a PDMA transfer unloads the entire packet to processor.

3.3.2 Transmitting

Similarly, PDMA also benefits the data loading process. User can initiate a PDMA transfer to load the entire packet into TX fifo. Note that user can start TX transmit without waiting the DMA transfer completed. Once the TX fifo is empty, a TX complete interrupt asserts. The signal indicates user is able to load the next packet. Figure 3.5 shows the interrupts

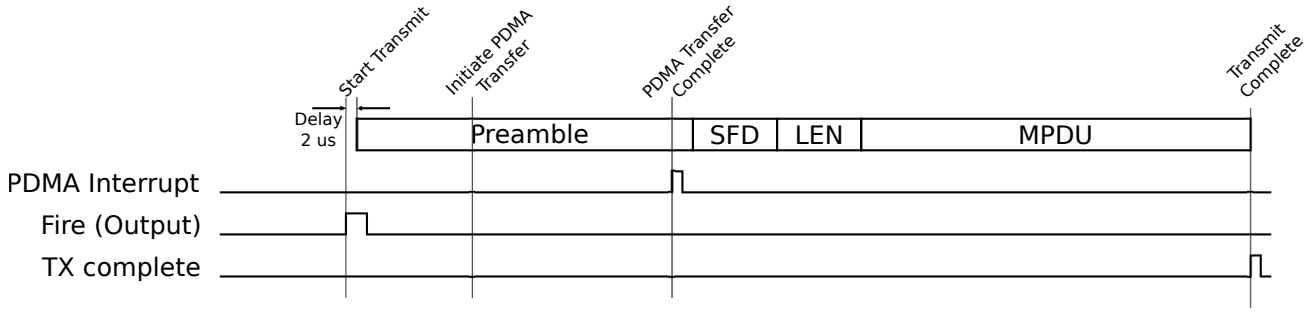


Figure 3.5: Interrupts activity during transmitting

activity. In this example, "Fire" is a MSS GPIO output which connects to the fabric. User asserts Fire to start transmitting the packet. It is not necessary to upload the packet before Fire assert. However, data must be start loading before SFD has been transmitted.

[3] In 802.15.4 standard, the length field should be greater or equal to 5 and less or equal to 127

3.4 Drivers

3.4.1 max2831

The following functions are provided in the max2831 driver. Although the MAX2831 internal registers' cannot be read through SPI interface, the driver maintains the registers' values. User can retrieves these value through a provided function.

1. void initialize_chip();
2. void setToDefaultRX(int register_number);
3. void setRegisterValueRX(int register_number,uint32_t register_value);
4. uint32_t getRegisterValueRX(int reg_number);
5. uint8_t setLNAGain(uint8_t LNAGain);
6. uint8_t setRXBaseBandVGAGain(uint8_t VGAGain);
7. uint8_t setTXBaseBandVGAGain(uint8_t VGAGain);
8. void enRXParVGAGainCtl();
9. uint8_t setFreqDivider(int freq);
10. int getFreqDivider();
11. uint8_t setBaseBandLowPassFilterMode (uint8_t LPFMode);
12. uint8_t setTXBaseBandLowPassFilterModeFineAdjust (uint8_t LPFMode, uint8_t LPFFineAdjust);
13. uint8_t setRXBaseBandLowPassFilterModeFineAdjust (uint8_t LPFMode, uint8_t LPFFineAdjust);

void initialize_chip()

Input argument: None
Return value: None
This function resets all registers to the default value.

void setToDefaultRX(int register_number)

Input argument: Register number. A valid register number is from 0 to 15.
Return value: None
This function sets the specified register to its default value.

void setRegisterValueRX(int register_number,uint32_t register_value)

Input argument: Register number, Register value. A valid register number is from 0 to 15.
Return value: None
This function set the specified register to specified value.

uint32_t getRegisterValueRX(int reg_number)

Input argument: Register number. A valid register number is from 0 to 15.
Return value: Register value
This function returns the specified register's value.

uint8_t setLNAGain(uint8_t LNAGain)

Input argument: LNA gain. A valid LNA gain is from 0 to 3.
Return value: Success/Fail. 0: Fail, 1: Success.
This function sets the LNA gains in RX loop and return success/fail.

LNA Gain	Mode
0 or 1	Low gain
2	Medium gain
3	High gain

Table 3.1: LNA gain mode

uint8_t setRXBaseBandVGAGain(uint8_t VGAGain)

Input argument: VGA gain. A valid VGA gain is from 0 (min) to 31 (max). Step size = 2 dB
Return value: Success/Fail. 0: Fail, 1: Success.
This function sets the **RX** baseband gain.

uint8_t setTXBaseBandVGAGain(uint8_t VGAGain)

Input argument: VGA gain. A valid VGA gain is from 0 (min) to 63 (max). Step size = 2 dB
Return value: Success/Fail. 0: Fail, 1: Success.
This function set the **TX** baseband gain.

void enRXParVGAGainCtl()

Input argument: None
Return value: None
This function enables the parallel control over the RX gain. It's necessary for AGC and Only valid on μSDR v2.

uint8_t setFreqDivider(int freq)

Input argument: RF Frequency
Return value: Success/Fail. 0: Fail, 1: Success.
This function sets the RF frequency of MAX2831.

int getFreqDivider()

Input argument: None
Return value: RF frequency This function returns the RF frequency of MAX2831.

uint8_t setTXBaseBandLowPassFilterModeFineAdjust (uint8_t LPFMode, uint8_t LPFFineAdjust)

Input argument: Low Pass Filter Mode, Adjustment
Return value: Success/Fail. 0: Fail, 1: Success.
See table 3.2 for more details.

uint8_t setRXBaseBandLowPassFilterModeFineAdjust (uint8_t LPFMode, uint8_t LPFFineAdjust)

Input argument: Low Pass Filter Mode, Adjustment
Return value: Success/Fail. 0: Fail, 1: Success.
See table 3.2 for more details.

<table>
<tr><td rowspan="3">LPF Mode</td><td colspan="11">LPF Fine Adjustment</td></tr>
<tr><td colspan="5">RX</td><td colspan="6">TX</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr>
<tr><td>0</td><td rowspan="4">90%</td><td rowspan="4">95%</td><td>7.5 MHz</td><td rowspan="4">105%</td><td rowspan="4">110%</td><td rowspan="4">90%</td><td rowspan="4">95%</td><td>8 MHz</td><td rowspan="4">105%</td><td rowspan="4">110%</td><td rowspan="4">115%</td></tr>
<tr><td>1</td><td>8.5 MHz</td><td>11 MHz</td></tr>
<tr><td>2</td><td>15 MHz</td><td>16.5 MHz</td></tr>
<tr><td>3</td><td>18 MHz</td><td>22.5 MHz</td></tr>
</table>

Table 3.2: LPF adjustment details

3.4.2 tx_packet

The **tx_packet_str** is a abstraction of a TX packet structure. Most of the functions are used to set the FCF field in TX packet. FCF has two bytes in length and each bits represent different information. Figure 3.6 gives a brief view of FCF field. A tx packet contains header and payloads. **tx_packet_str** stores most of the header information except the **source address**, **source address mode** and **source PAN ID**. These information can be retrieved by a pointer to a radio (**rp**) structure which owns this tx packet. Also, **data_ptr** is the pointer to the payload.

15-14	13-12	11-10	9-8
Source Address Mode	Reserved	Destination Address Mode	Reserved

7	6	5	4	3	2-0
Reserved	Intra PAN	ACK Request	Frame Pending	Security Enable	Frame Type

Figure 3.6: Detail information of FCF field

Listing 3.1: TX Packet Structure

```
struct tx_packet_str{
        uint8_t frame_type;
        uint8_t security_en;
        uint8_t frame_pending;
        uint8_t ack_req;
        uint8_t intra_pan;
        uint8_t dest_addr_mode;
        uint8_t dsn;
        uint16_t dest_pan_ID;
        uint32_t dest_addr_LSB, dest_addr_MSB;
        uint8_t pkt_overhead;

        uint8_t data_length;
        uint32_t* data_ptr;
        struct radio* rp;
};
```

1. inline uint8_t set_dest_addr_mode(struct tx_packet_str* tp, uint8_t dam);
2. inline void set_pan_id_comp(struct tx_packet_str* tp, uint8_t ip);
3. inline void set_ack(struct tx_packet_str* tp, uint8_t ar);
4. inline void set_frame_pending(struct tx_packet_str* tp, uint8_t fp);
5. inline void set_security(struct tx_packet_str* tp, uint8_t se);
6. inline void set_frame_type(struct tx_packet_str* tp, uint8_t ft);
7. inline void set_DSN(struct tx_packet_str* tp, uint8_t DSN);
8. inline void set_dest_pan(struct tx_packet_str* tp, uint16_t dp);
9. inline void set_dest_addr(struct tx_packet_str* tp, uint32_t da1, uint32_t da0);
10. void calculate_FCF(struct tx_packet_str* tp, uint32_t* tx_pkt_ptr);
11. uint8_t calculate_address(struct tx_packet_str* tp, uint32_t* tx_pkt_ptr);
12. uint8_t data_trans (struct tx_packet_str* tp);

Source/Destination Address Mode	Address Field Length
0	No PAN/Address present
2	2 Bytes
3	8 Bytes

Table 3.3: List of destination address mode

inline uint8_t set_dest_addr_mode(struct tx_packet_str* tp, uint8_t dam)

Input argument: tx_packet_str pointer, destination address mode (dam).
Return value: Success/Fail. 0: Fail, 1: Success.

inline void set_pan_id_comp(struct tx_packet_str* tp, uint8_t ip)

Input argument: tx_packet_str pointer, Intra PAN (ip)
Return value: None
"The PAN ID Compression subfield is 1 bit in lengthand specifies whether the MAC frame is to be sent containing only one of the PAN identifier fields whenboth source and destination addresses are present" [2] The available options are following:

Intra PAN	Dest. PAN	Dest. Addr.	Source PAN	Source Addr.
1	Present	Present	None [a]	Present
0	Present	Present	Present	Present
0	Present	Present	None [b]	None [b]
0	None [b]	None [b]	Present	Present
0	None [b]	None [b]	None [b]	None [b]

[a] Source PAN ID. equals to Destination PAN ID.
[b] Refer table 3.3 for detail information.

Table 3.4: PAN ID compression and source/destination address mode

inline void set_ack(struct tx_packet_str* tp, uint8_t ar)

Input argument: tx_packet_str pointer, ACK request (ar)
Return value: None
Note that the ACK request bit should not be set in broadcast [4] packet.

ACK request (ar)	Effect
0	No ACK
1	Request ACK

Table 3.5: ACK Request

inline void set_frame_pending(struct tx_packet_str* tp, uint8_t fp)

Input argument: tx_packet_str pointer, frame pending(fp)
Return value: None

inline void set_security(struct tx_packet_str* tp, uint8_t se)

Input argument: tx_packet_str pointer, Security Enable (se)
Return value: None

[4] Destination address 0xffff or 0xffffffffffffffff

frame pending (fp)	Effect
0	No frame pending
1	Frame pending

Table 3.6: Frame Pending

Security Enable (se)	Effect
0	Security Enable
1	Security Disable

Table 3.7: Security enable

inline void set_frame_type(struct tx_packet_str* tp, uint8_t ft)

Input argument: tx_packet_str pointer, Frame Type (ft)
Return value: None

Frame Type (ft)	Effect
0	Beacon
1	Data
2	Acknowledgment
3	Mac Command

Table 3.8: Frame type

inline void set_DSN(struct tx_packet_str* tp, uint8_t DSN)

Input argument: tx_packet_str pointer, Data Sequence Number (DSN)
Return value: None
This function sets the DSN field in a TX packet.

inline void set_dest_pan(struct tx_packet_str* tp, uint16_t dp)

Input argument: tx_packet_str pointer, Destination PAN ID (dp)
Return value: None
This function sets the destination PAN ID in a TX packet.

inline void set_dest_addr(struct tx_packet_str* tp, uint32_t da1, uint32_t da0)

Input argument: tx_packet_str pointer, Destination Address (da1, da0)
Return value: None
This function sets the destination address in a TX packet. Destination address could be as long as 8 bytes (destination address mode = 3). If 8 bytes mode is selected, destination address = da1 (higher 32-bits), da0 (lower 32-bits). Similarly, destination address = (da0 & 0xffff) if 2 bytes mode is selected.

void calculate_FCF(struct tx_packet_str* tp, uint32_t* tx_pkt_ptr)

Input argument: tx_packet_str pointer, TX packet header pointer
Return value: None
This function aggregates all the settings to 2 bytes FCF field.

uint8_t calculate_address(struct tx_packet_str* tp, uint32_t* tx_pkt_ptr)

Input argument: tx_packet_str pointer, TX packet header pointer
Return value: This function calculates address format in a TX packet by source/destination address mode and Intra PAN ID bit.

uint8_t data_trans (struct tx_packet_str* tp)

Input argument: tx_packet_str pointer
Return value: Success/Fail. 0: Fail, 1: Success.
This function initiates a PDMA transfer which load the packet into TX fifo.

3.4.3 rx_packet

This driver is a "reverse" operation of tx_packet. The driver extracts the parameters in header from a received packet. The following source code is the **rx_packet_str** structure. It's very similar to tx_packet_str. The major diferences are the source address information, rssi value and crc result.

Listing 3.2: RX Packet Structure

```
struct rx_packet_str{
        uint8_t frame_type;
        uint8_t security_en;
        uint8_t frame_pending;
        uint8_t ack_req;
        uint8_t intra_pan;
        uint8_t dest_addr_mode;
        uint8_t dsn;
        uint16_t dest_pan_ID;
        uint32_t dest_addr_LSB, dest_addr_MSB;

        uint8_t packet_length;
        uint32_t payload_idx;

        uint8_t src_addr_mode;
        uint16_t src_pan_ID;
        uint32_t src_addr_LSB, src_addr_MSB;
        uint8_t rssi;
        uint8_t crc;
};
```

1. void read_fifo(uint32_t* rdata);
2. inline uint8_t isFifoEmpty();
3. uint8_t rx_packet_create(struct rx_packet_str* rp, uint32_t* rx_data);

void read_fifo(uint32_t* rdata)

Input argument: A pointer where stores the unloaded data.
Return value: None
This function unload **one packet** [5] from RX fifo. The unloaded data will be stored in **rdata** array. The size of rdata cannot be too small to store a single packet.

inline uint8_t isFifoEmpty()

Input argument: None
Return value: Fifo empty
This function checks whether the RX fifo is empty or not. If it's empty, function returns 1 and vice versa.

uint8_t rx_packet_create(struct rx_packet_str* rp, uint32_t* rx_data)

Input argument: rx_packet_str pointer, a pointer where stores the unloaded data.
Return value: Success/Fail. 0: Fail, 1: Success.
This function extracts and stores all the data member in rx_packet_str from a given rx_data pointer.

[5] Only the oldest packet in RX fifo.

3.4.4 radio_config

This driver provides the low-level radio hardware configuration and source address information. The radio structure below shows its data member. All the data member are address correlated. User has to note that the fabric doesn't perform the address recognition because of the area constraint. In order for the radio to operate properly, user has to perform address recoginition in time. In addition, multiple source addresses is supported by software configuration.

Listing 3.3: Radio Structure

```
struct radio{
        uint8_t multiple_addr_en;
        uint8_t num_of_multiple_addr;
        uint32_t* multiple_addr_ptr;
        uint16_t src_pan_ID;
        uint8_t src_addr_mode;
        uint32_t src_addr_LSB, src_addr_MSB;
};
```

1. void init_system();
2. inline void tx_fire();
3. void RF_shdn(uint8_t shdn);
4. inline void auto_ack_en(uint8_t ack_en);
5. inline void rx_agc_en(uint8_t agc_mode);
6. inline void ack_flush();
7. inline void auto_tx_en(uint8_t en);
8. uint8_t get_radio_mode();
9. uint8_t dest_addr_filter(struct rx_packet_str* rp, struct radio* r0);
10. void set_multiple_address(struct radio* r0, uint8_t en, uint8_t nu, uint32_t* map);
11. inline uint8_t set_src_addr_mode(struct radio* r0, uint8_t sam);
12. inline void set_src_pan(struct radio* r0, uint16_t sp);
13. inline void set_src_addr(struct radio* r0, uint32_t sa1, uint32_t sa0);

void init_system()

Input argument: None
Return value: None
This function initialize the entire radio including MAX2831, GPIO and PDMA drivers. This function should be called before any other radio correlated functions.

inline void tx_fire()

Input argument: None
Return value: None
This function starts transmitting the content of TX fifo.

void RF_shdn(uint8_t shdn)

Input argument: radio shutdown
Return value: None
This function turns off all the radio peripherals. (RF, ADCs and DAC)

inline void auto_ack_en(uint8_t ack_en)

Input argument: ACK enable/disable
Return value: None

Auto ACK en (ack_en)	Effect
0	Disable Auto ACK
1	Enable Auto ACK

Table 3.9: ACK Enable/Disable

inline void rx_agc_en(uint8_t agc_mode)

Input argument: AGC mode
Return value: None

AGC mode (agc_mode)	Effect
0	Disable AGC
1	SFD-Latch AGC
2	Reserved
3	Continues AGC

Table 3.10: AGC Mode

inline void ack_flush()

Input argument: None
Return value: None
This function flushes the outgoing ACK packet. Once a packet reception completed, user should unload the packet from the radio and perform destination address filter. It returns whether the address is match to radio's source address or not. In the mean time, user needs to check the coming packet has ACK request bit set or not. If the following conditions are met, user should call ack_flush();

1. Destination address **doesn't** match to local source address.
2. Packet passes the CRC check.
3. Packet has ACK request field set.

inline void auto_tx_en(uint8_t en)

Input argument: Auto TX enable
Return value: None
This function enables the automatic transmission of TX fifo. User can achieve concurrent transmission using this function.

Auto TX enable (en)	Effect
0	Disable Auto TX
1	Radio starts transmitting after 192 μs of any packet received

Table 3.11: Auto TX

uint8_t get_radio_mode()

Input argument: None
Return value: Radio mode
This function is mainly used for debugging purpose.

uint8_t dest_addr_filter(struct rx_packet_str* rp, struct radio* r0)

Input argument: rx_packet_str pointer, radio struct pointer
Return value: Match/Mismatch. 0: Mismatch, 1: Match.

Radio Mode	Fabric State	Corresponding State
0x00	RX_IDLE	Idle Listening
0x01	RX_COLLECT	RX
0x03	TX_RX_TURNAROUND	
0x04	RX_TX_TURNAROUND	
0x05	RX_WAIT_FOR_ACK_GLOSSY	
0x08	RADIO_OFF	Off
0x09	RADIO_WARMUP	
0x0c	WAIT_FOR_TX_COMPLETE	TX
0x0d	RX_WAIT_FOR_ACK_GLOSSY_COMPLETE	

Table 3.12: Radio Mode

void set_multiple_address(struct radio* r0, uint8_t en, uint8_t nu, uint32_t* map)

Input argument: radio struct pointer, multiple address enable, number of multiple address, multiple address array
Return value: None

inline uint8_t set_src_addr_mode(struct radio* r0, uint8_t sam)

Input argument: radio struct pointer, Source Address Mode (sam)
Return value: Success/Fail. 0: Fail, 1: Success.
See table 3.3 for detail information.

inline void set_src_pan(struct radio* r0, uint16_t sp)

Input argument: radio struct pointer, Source PAN ID (sp)
Return value: None
This function sets the source PAN ID in a radio struct.

inline void set_src_addr(struct radio* r0, uint32_t sa1, uint32_t sa0)

Input argument: radio struct pointer, Source Address (sa1, sa0)
Return value: None
This function sets the source address in a radio struct. Source address could be as long as 8 bytes (source address mode = 3). If 8 bytes mode is selected, source address = sa1 (higher 32-bits), sa0 (lower 32-bits). Similarly, source address = (sa0 & 0xffff) if 2 bytes mode is selected.

3.5 Example Codes (LED counter)

Listing 3.4: Initialize system

```
RF_shdn(0);
init_system();
res = setTXBaseBandVGAGain(10);
uint32_t frequency = 2405;
res = setFreqDivider(frequency);
rx_agc_en(3);
```

Listing 3.5: Create object

```
struct radio* radio0;
radio0 = malloc(sizeof(struct radio));
radio0->multiple_addr_en = 0;

struct tx_packet_str* led_pkt;
led_pkt = malloc(sizeof(struct tx_packet_str));
// link to radio0
led_pkt->rp = radio0;

struct rx_packet_str* received_packet;
received_packet = malloc(sizeof(struct rx_packet_str));
```

Listing 3.6: Set packet & Radio

```
uint32_t* led_data_ptr;
uint32_t data_size = 4;
led_data_ptr = malloc(sizeof(uint32_t)*data_size);
led_pkt->data_length = data_size;
led_pkt->data_ptr = led_data_ptr;

// set led packet
set_frame_type(led_pkt, 1);
set_security(led_pkt, 0);
set_pan_id_comp(led_pkt, 1);
set_dest_addr_mode(led_pkt, 2);
set_DSN(led_pkt, 0);
set_dest_pan(led_pkt, 0x22);
set_frame_pending(led_pkt, 0);

set_dest_addr(led_pkt, 0, 0xffff);
set_ack(led_pkt, 0);

// set radio
set_src_pan(radio0, 0x22);
set_src_addr_mode(radio0, 2);
set_src_addr(radio0, 0, 0x05);

// set radio packet data
led_data_ptr[0] = 0x3f;
led_data_ptr[1] = 0x06;
led_data_ptr[2] = 0x00;
led_data_ptr[3] = 0x00;
```

Listing 3.7: Periodic transmit

```
if (TIM1_int & PDMA_int){
        tx_fire();
        PDMA_int = 0;
        TIM1_int = 0;
}
```

Listing 3.8: Load packet

```
if (tx_complete_int==1){
        tx_complete_int = 0;
        tx_counter++;
        led_data_ptr[2] = (tx_counter & 0xff00)>>8;
        led_data_ptr[3] = (tx_counter & 0xff);
        set_DSN(led_pkt, led_data_ptr[3]);
        data_trans (led_pkt);
}
```

Listing 3.9: Unload packet

```
if (rx_pkt_done_int==1){
        rx_pkt_done_int = 0;
        read_fifo(rdata);

        res = rx_packet_create(received_packet, rdata);

        uint8_t crc_correct, ack_req;
        crc_correct = received_packet->crc;
        ack_req = received_packet->ack_req;
        if (crc_correct){
                uint32_t* rx_data = rdata + received_packet->payload_idx;

                uint8_t address_match = dest_addr_filter(received_packet, radio0);

                // address doesn't match but requests ACK
                // flush ack
                if ( (!address_match) & ack_req){
                        ack_flush();
                }

                // broadcast
                if (address_match==3){
                // led packet signature, 0x3f, 0x06
                        if ((rx_data[0]==0x3f)&&(rx_data[1]==0x06)){
                                set_led(rx_data[3]);
                        }
                }

        }

}
```

3.6 Document Revision History

- Revision 0.2 (rxxxx) – Feb 24, 2013

  English/grammar corrections

  Adding notes on todo items

- Revision 0.1 (r1525) – Sep 26, 2012

  Initial revision

Bibliography

[1] Analog Device. AD9288: 8-Bit, 40/80/100 MSPS Dual A/D Converter. http://www.analog.com/static/imported-files/data_sheets/AD9288.pdf.

[2] IEEE Computer Society. Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs).

[3] Maxim. MAX2831: 2.4GHz to 2.5GHz 802.11g/b RF Transceivers with Integrated PA. http://datasheets.maxim-ic.com/en/ds/MAX2831-MAX2832.pdf.

[4] Maxim. MAX5189: Dual, 8-Bit, 40MHz, Current/Voltage, Simultaneous-Output DACs. http://datasheets.maxim-ic.com/en/ds/MAX5186-MAX5189.pdf.

[5] MAXIM. Precision, Micropower, Low-Dropout, High-Output-Current, SOT23 Voltage References. http://datasheets.maximintegrated.com/en/ds/MAX6061-MAX6068.pdf.

[6] NXP. Integrated RTC, TCXO and quartz crystal. http://www.nxp.com/documents/data_sheet/PCF2127A.pdf.

[7] Silvertel. AG9200-S Power-over-Ethernet Module. http://www.silvertel.com/DataSheets/Ag9200v1-7.pdf.

[8] SKYWORKS. AS213-92: PHEMT GaAs IC SPDT Switch 0.1-3 GHz. http://www.skyworksinc.com/uploads/documents/200193D.pdf.

[9] Texas Instruments. ADC081S101: Single Channel, 0.5 to 1 Msps, 8-Bit A/D Converter. http://www.ti.com/lit/ds/symlink/adc081s101.pdf.