SOFTWARE MANUAL

AXRADIO API Manual

Version V5

20210127

ON Semiconductor®

1. TABLE OF CONTENTS

2. Introduction

This document describes axradio. Axradio is intended to be an easy to use "driver" for OnSemi Radio Chips, such as the AX5043, AX5045 allowing the user to focus on his product functionality and not the details of the radio link.

The API consists of a header file, axradio.h, providing functions for transmitting and receiving packets, and switching the radio into different modes.

Most of the transceiver configuration is computed by AXRadioLab and stored in non-volatile memory; it will not be changeable by the firmware. These items include:

- Radio PHY configuration (except for channel)
- Radio MAC (Frame Format) configuration (except actual addresses)

Some items should be runtime configurable. Items include:

- Channel Number
- Actual MAC Addresses

3. Prerequisites

AXRadio uses the Wakeup Timer facilities of LibMF (libmfwtimer.h). They provide:

- 2 Timers, a slow but persistently running timer for events that need to survive sleep, and a fast but switched off during sleep timer for short intervals
- Multiple events per timer are supported
- Not only usable for AXRadio – also supports user events
- Mechanism for scheduling callbacks from interrupt context which run in main context
- Less overhead (assembly implementation where C compiler generated code was too inefficient)

In order for the LibMF Wakeup Timer to work properly, the user should not use enter_sleep and enter_standby directly, but should use wtimer_idle instead. Also, during long running computations, wtimer_idle should be called periodically to allow pending callbacks to run.

4. Constants

4.1. Operating Modes

Mode	Description
AXRADIO_MODE_OFF	The radio is off, but in a configured state that allows it to be switched quickly to any other mode
AXRADIO_MODE_DEEPSLEEP	The radio is in deep sleep mode; this mode has the lowest possible power consumption, but may take some time to leave
AXRADIO_MODE_CW_TRANSMIT	The radio is setup for CW (constant carrier wave) transmit (used for basic tests)
AXRADIO_MODE_ASYNC_TRANSMIT	The radio is set up for asynchronous transmission. After a call to axradio_transmit, the transmitter is switched on and the packet is sent, afterwards the transmitter is switched off again.
AXRADIO_MODE_WOR_TRANSMIT	Same as AXRADIO_MODE_ASYNC_TRANSMIT, but starts transmission with a wakeup preamble
AXRADIO_MODE_ACK_TRANSMIT	The radio is set up for asynchronous transmission. After a call to axradio_transmit, the transmitter is switched on and the packet is sent, afterwards the radio waits for an acknowledge from the slave. If no acknowledge is received within a configured timespan, the packet is (optionally) retransmitted.
AXRADIO_MODE_WOR_ACK_TRANSMIT	Same as AXRADIO_MODE_ACK_TRANSMIT, but starts transmission with a wakeup preamble
AXRADIO_MODE_ASYNC_RECEIVE	The radio is setup for continuous asynchronous receive.
AXRADIO_MODE_WOR_RECEIVE	The radio is setup for wake-on-radio receive.
AXRADIO_MODE_ACK_RECEIVE	The radio is setup for continuous asynchronous receive. After a packet is received, an acknowledge packet is sent back.
AXRADIO_MODE_WOR_ACK_RECEIVE	The radio is setup for wake-on-radio receive. After a packet is received, an acknowledge packet is sent back.
AXRADIO_MODE_STREAM_TRANSMIT	The radio is setup for streaming transmit (used for basic tests)
AXRADIO_MODE_STREAM_TRANSMIT_UNENC	The radio is setup for unencoded streaming transmit (used for basic tests)
AXRADIO_MODE_STREAM_TRANSMIT_SCRAM	The radio is setup for scrambled streaming transmit (used for basic tests)
AXRADIO_MODE_STREAM_RECEIVE	The radio is setup for streaming receive (used for basic tests)
AXRADIO_MODE_STREAM_RECEIVE_UNENC	The radio is setup for unencoded streaming receive (used for basic tests)

Mode	Description
AXRADIO_MODE_STREAM_RECEIVE_SCRAM	The radio is setup for scrambled streaming receive (used for basic tests)
AXRADIO_MODE_SYNC_MASTER	The radio is setup as synchronous master.
AXRADIO_MODE_SYNC_ACK_MASTER	The radio is setup as synchronous master. The master expects an acknowledge from the slave, and reports the acknowledge or the absence of an acknowledge to the caller.
AXRADIO_MODE_SYNC_SLAVE	The radio is setup as synchronous slave.
AXRADIO_MODE_SYNC_ACK_SLAVE	The radio is setup as synchronous slave. The slave sends an acknowledge whenever a packet is received.

4.2. Error Codes

Error Code	Description
AXRADIO_ERR_NOERROR	No error occurred, operation completed successfully
AXRADIO_ERR_NOTSUPPORTED	The operation is not supported
AXRADIO_ERR_BUSY	The operation could not be completed because the radio was busy
AXRADIO_ERR_TIMEOUT	The operation timed out (eg. the maximum number of retransmission exceeded)
AXRADIO_ERR_INVALID	The operation failed because of an invalid parameter
AXRADIO_ERR_NOCHIP	No radio chip was found
AXRADIO_ERR_RANGING	The frequency could not be ranged
AXRADIO_ERR_LOCKLOST	PLL lock was lost
AXRADIO_ERR_RETRANSMISSION	This packet is a retransmission (due to no acknowledge received)
AXRADIO_ERR_RESYNC	The synchronous slave restarts resynchronization.
AXRADIO_ERR_RESYNCTIMEOUT	The synchronous slave pauses resynchronization.

Error Code	Description
AXRADIO_ERR_RECEIVESTART	The synchronous slave powers up the receiver.

4.3. Status Change Codes

Status Change Code	Description
AXRADIO_STAT_RECEIVE	Receive Packet arrived
AXRADIO_STAT_TRANSMITSTART	Transmitter start notification
AXRADIO_STAT_TRANSMITDATA	Transmitter new data needed notification for streaming transmit modes
AXRADIO_STAT_TRANSMITEND	Transmitter end notification
AXRADIO_STAT_RECEIVESFD	Receiver SFD detected notification
AXRADIO_STAT_CHANNELSTATE	Channel state update

In the acknowledge modes, at the start of transmission, TRANSMITSTART is called either with NOERROR or RETRANSMISSION, depending on whether it is the first or subsequent transmission of a packet. At the end of the packet transmission, TRANSMITEND is called with BUSY. When an acknowledgement is received, TRANSMITEND is called again with NOERROR. If no acknowledgement is received after a timeout, and the number of retransmissions is used up, TRANSMITEND is called with TIMEOUT.

The synchronous master first calls TRANSMITDATA approximately 1ms before turning on the transmitter. This call may be used to prepare the transmit packet and call axradio_transmit.

5. Functions

5.1. uint8_t axradio_init(void)

Initialize the driver and the chip. This routine must be called before any other axradio routine is called. Returns one of the following error codes:

Error Code	Description
AXRADIO_ERR_NOERROR	No error occurred.
AXRADIO_ERR_NOCHIP	No radio chip was found
AXRADIO_ERR_RANGING	The frequency could not be ranged

5.2. uint8_t axradio_cansleep(void)

This function returns one if the processor can go to sleep without adverse effects, or zero if the required interrupt latency could not be fulfilled when going to sleep.

This function should be used as follows:

```
wtimer_runcallbacks();
uint8_t flags = WTFLAG_CANSTANDBY;
if (axradio_cansleep())
        flags |= WTFLAG_CANSLEEP;
wtimer_idle(flags);
```

5.3. uint8_t axradio_set_mode(uint8_t mode)

This function sets the mode of the radio. Supply one of the AXRADIO_MODE_* constants. Not all modes may be supported, depending on the configuration set in AXRadioLab.

It returns one of the following error codes:

Error Code	Description
AXRADIO_ERR_NOERROR	No error occurred, operation completed successfully
AXRADIO_ERR_NOTSUPPORTED	The operation is not supported
AXRADIO_ERR_NOCHIP	No radio chip was found
AXRADIO_ERR_RANGING	The frequency could not be ranged

5.4. UINT8_T AXRADIO_GET_MODE(VOID)

This function returns the current chip operating mode. See the AXRADIO_MODE_* constants.

5.5. UINT8_T AXRADIO_SET_CHANNEL(UINT8_T CHNUM)

This function sets the channel number to be used. The mapping between channel number and frequency is configured in AXRadioLab. This function returns one of the following error codes:

Error Code	Description
AXRADIO_ERR_NOERROR	No error occurred, operation completed successfully
AXRADIO_ERR_BUSY	The operation could not be completed because the radio was busy
AXRADIO_ERR_INVALID	The operation failed because of an invalid parameter
AXRADIO_ERR_RANGING	The frequency could not be ranged

5.6. UINT8_T AXRADIO_GET_CHANNEL(VOID)

This function returns the currently used channel number.

5.7. UINT8_T AXRADIO_GET_PLLRANGE(VOID) UINT8_T AXRADIO_GET_PLLRANGE_TX(VOID)

These functions return the current PLL ranges for the currently set frequency. This is mainly for debugging.

5.8. VOID AXRADIO_SET_LOCAL_ADDRESS(CONST STRUCT AXRADIO_ADDRESS_MASK *ADDR)

This function sets the MAC address of the local radio node. The length of a MAC address is configured by AXRadioLab.

The axradio_address_mask structure has the following definition:

```
struct axradio_address_mask {
    uint8_t addr[4];
    uint8_t mask[4];
};
```

5.9. void axradio_get_local_address(struct axradio_address_mask *addr)

This function returns the currently configured local radio node MAC address. A pointer to a memory space where the address can be stored into must be provided.

5.10. uint8_t axradio_set_freqoffset(int32_t offs)

Axradio allows the user to shift the transceiver somewhat from the channel center frequency. This can be useful in a master-slave setup, where the slaves adjust their frequency upon reception from the master (axradio measures the frequency offset of every packet received), to compensate for drifting crystals.

This routine sets the frequency offset from the channel center frequency that should be used. The offset remains when channels are switched.

The unit is a driver internal one. It can be converted to and from Hz using the axradio_conv_freq_* routines.

5.11. int32_t axradio_get_freqoffset(void)

This routine returns the current frequency offset.

5.12. int32_t axradio_conv_freq_tohz(int32_t f)

This routine convert’s internal unit frequency offsets to Hz.

5.13. int32_t axradio_conv_freq_fromhz(int32_t f)

This routine converts frequency offsets in Hz into internal units for frequency offset.

5.14. int32_t axradio_conv_timeinterval_totimer0(int32_t dt)

This function converts a time interval, such as a difference of two status callback st->time values, into wakeup timer 0 units.

5.15. uint32_t axradio_conv_time_totimer0(uint32_t dt)

This function converts an absolute time, such as a status callback st->time value, from internal units to wakeup timer 0 units. Note that status callback st->time values are generally only valid during the status callback, as the relationship between the internal timer and the wakeup timer 0 may change, for example when the radio chip is powered down.

5.16. void axradio_calibrate_lposc(void);

This function calibrates the Radio's low power 640Hz RC oscillator from a given TCXO 48MHz reference frequency. This function should be invoked when the device is operated in Wake on Radio mode only.

5.17. UINT8_T AXRADIO_TRANSMIT(CONST STRUCT AXRADIO_ADDRESS *ADDR, CONST UINT8_T *PKT, UINT16_T LEN)

Calling this function transfers the user packet data pointed to by pkt and having length len to axradio for transmission. Only one packet may be in the process of being transmitted at any time. If a second packet transmission is attempted, a busy error is returned.

The semantics of this routine slightly differs depending on whether the driver is in an asynchronous or a synchronous mode.

In an asynchronous mode, calling this routine queues the packet and immediately starts transmission.

In a synchronous mode, the data is stored for transmission in the next time slot. If this routine is called a second time before the next time slot, the old data is replaced by the data passed in the second call. This may be used to record default data early in the cycle, and possibly update the data if something happens.

The addr argument specifies the address of the remote station this packet is destined to.

The axradio_address structure has the following definition:

```
struct axradio_address {
        uint8_t addr[4];
};
```

5.18. UINT8_T AXRADIO_AGC_FREEZE(VOID) UINT8_T AXRADIO_AGC_THAW(VOID)

axradio_agc_freeze/axradio_agc_thaw may be used during the streaming receive modes to freeze or thaw the automatic gain control.

5.19. VOID AXRADIO_STATUSCHANGE(CONST __XDATA STRUCT AXRADIO_STATUS *ST)

This function must be provided by the user code. It is called by axradio whenever an event that needs to be notified happens.

```
struct axradio_status {
        uint8_t status; // one of the AXRADIO_STAT_* constants
        uint8_t error; // one of the AXRADIO_ERR_* constants
        uint32_t time; // timestamp of the event
        //
        union {
                // status == AXRADIO_STAT_RECEIVE
                struct {
```

```
                        struct {
                                int8_t rssi; // RSSI, dBm
                                int32_t offset; // frequency offset, internal units
                        } phy;
                        struct {
                                uint8_t remoteaddr[4];
                                uint8_t localaddr[4];
                                const __xdata uint8_t *raw;
                        } mac;
                        const __xdata uint8_t *pktdata;
                        uint16_t pktlen;
                } rx;
                // status == AXRADIO_STAT_CHANNELSTATE
                struct {
                        int8_t rssi; // RSSI, dBm
                        uint8_t busy; // 1=over the LBT threshold
                } cs;
        } u;
};
```

6. Static Configuration Items

These static configuration constants are computed by AX-RadioLab.

6.1. extern const __code uint8_t axradio_maclen;

This constant contains the length of the MAC header in front of the user packet data.

6.2. extern const __code uint8_t axradio_addrlen;

This constant contains the length of a MAC address, and may be in the range of 1–4.

7. Example usage code

This section lists simplified code to illustrate the usage of the API.

The AX Radiolab demonstrates firmware examples on devices namely AX8052F143/ AX8052F145 and AXM0F243 as below.

7.1. Simple Asynchronous

This example shows the skeleton for the simplest possible asynchronous transmitter to receiver case.

7.1.1. Transmitter

AX8052F143/ AX8052F145 Devices

```
#include "ax8052.h"
#include "libmftypes.h"
#include "libmfwtimer.h"
#include "libmfflash.h"
#include "libmfradio.h"
#include "axradio.h"

static const __code struct axradio_address_mask localaddr = {
    { 0x12, 0x34, 0x56, 0x78 },
    { 0xFF, 0xFF, 0xFF, 0xFF }
};

static const __code struct axradio_address remoteaddr = {
    { 0xCA, 0xFE, 0xBA, 0xBE }
};

void axradio_statuschange(const __xdata struct axradio_status *st)
{
}

uint8_t _sdcc_external_startup(void)
{
    // initialize GPIO, peripherals
    if (PCON & 0x40)
        return 1;
    return 0;
```

```
}

#if defined(SDCC)
extern uint8_t _start__stack[];
#endif

void main (void)
{
#if !defined(SDCC)
    _sdcc_external_startup();
#else
    __asm
    G$_start__stack$0$0 = __start__stack
    .globl G$_start__stack$0$0
    __endasm;
#endif

    flash_apply_calibration(); // check for non-existing calibration
    CLKCON = 0x00;
    wtimer_init(CLKSRC_LPOSC, 1, CLKSRC_FRCOSC, 7);

    EA = 1;
    if (!(PCON & 0x40)) {
        axradio_init(); // check for error
        axradio_set_local_addr(&localaddr);
    } else {
        axradio_commsleepexit();
    }

    for (;;) {
        uint8_t flg;
        if (key is pressed) {
            __xdata uint8_t userpkt[..];
            // fill userpkt
            axradio_transmit(&remoteaddr, userpkt, sizeof(userpkt));
        }
        wtimer_runcallbacks();
        flg = WTFLAG_CANSTANDBY;
        if (axradio_cansleep())
```

```
            flg |= WTFLAG_CANSLEEP;
        wtimer_idle(flg);
    }
}
```

AXM0F243 device

```
void main (void)
{
        __enable_irq();

        #if defined USE_DBGLINK
            debuglink_init_axm0();              /// TODO
        #endif // USE_DBGLINK

    wtimer_init();
    if (coldstart)
    {
        axradio_setup_pincfg3();

        led0_off();
        led1_off();
        led2_off();
        led3_off();

        wakeup_desc.handler = wakeup_callback;
        i = axradio_init();
        axradio_set_local_address(&localaddr);
        axradio_set_default_remote_address(&remoteaddr);
       /* IMO calibration initialization and setup code start */
        /* Set Pin 0.0 (VTCXO pin) */
        GPIO_PRT0->DR_SET |= (1 << AXM0F2_VTCXO_PIN);

        /* IMO and ILO calibration setup */
        setup_osc_calibration(AXM0XX_HFCLK_CLOCK_FREQ, CLKSRC_RSYSCLK);
        /* IMO calibration initialization and setup code end */
        i = axradio_set_mode(RADIO_MODE);

    }//coldtsart
    else
    {
      axradio_commsleepexit();
      NVIC_EnableIRQ(GPIOPort2_IRQn);
            /* ENABLE RIRQ PR5 2.4 INTERRUPT */
      GPIO_PRT2->INTR_CFG |= 1 << 8;
```

```
    }

    for (;;) {
        uint8_t flg;
        wtimer_runcallbacks();
        crit = enter_critical();
        if(button_pressed) {
            exit_critical(crit);
            transmit_packet();
        }

        flg = WTFLAG_CANSTANDBY;
        if (axradio_cansleep())
            flg |= WTFLAG_CANSLEEP;
        wtimer_idle(flg);
    }
}
```

7.1.2. RECEIVER

AX8052F143/ AX8052F145 Devices

```
#include "ax8052.h"
#include "libmftypes.h"
#include "libmfwtimer.h"
#include "libmfflash.h"
#include "libmfradio.h"
#include "axradio.h"

static const __code struct axradio_address_mask localaddr = {
    { 0xCA, 0xFE, 0xBA, 0xBE },
    { 0xFF, 0xFF, 0xFF, 0xFF }
};

void axradio_statuschange(const __xdata struct axradio_status *st)
{
    switch (st->status) {
```

```
    case AXRADIO_STAT_RECEIVE:
        // check st->error
        // display st->u.rx.pktdata / st->u.rx.pktlen
        break;

    default:
        break;
    }
}

uint8_t _sdcc_external_startup(void)
{
    // initialize GPIO, peripherals
    if (PCON & 0x40)
        return 1;
    return 0;
}

#if defined(SDCC)
extern uint8_t _start__stack[];
#endif

void main (void)
{
#if !defined(SDCC)
    _sdcc_external_startup();
#else
    __asm
    G$_start__stack$0$0 = __start__stack
    .globl G$_start__stack$0$0
    __endasm;
#endif

    flash_apply_calibration(); // check for non-existing calibration
    CLKCON = 0x00;
    wtimer_init(CLKSRC_LPOSC, 1, CLKSRC_FRCOSC, 7);

    EA = 1;
    if (!(PCON & 0x40)) {
```

```
        axradio_init(); // check for error
        axradio_set_local_addr(&localaddr);
    } else {
        axradio_commsleepexit();
    }

    for (;;) {
        uint8_t flg;
        wtimer_runcallbacks();
        flg = WTFLAG_CANSTANDBY;
        if (axradio_cansleep())
            flg |= WTFLAG_CANSLEEP;
        wtimer_idle(flg);
    }
}
```

AXM0F243 device

```
#include <axm0f243.h>
#include "axm0_config.h"
#include "axm0f2_xbar.h"
#include "axm0f2.h"
#include <stdbool.h>
#include "axradio.h"

static const __code struct axradio_address_mask localaddr = {
    { 0xCA, 0xFE, 0xBA, 0xBE },
    { 0xFF, 0xFF, 0xFF, 0xFF }
};

void axradio_statuschange(const __xdata struct axradio_status *st)
{
    switch (st->status) {
    case AXRADIO_STAT_RECEIVE:
        // check st->error
        // display st->u.rx.pktdata / st->u.rx.pktlen
        break;

    default:
```

```
        break;
    }
}

/*This function is invoked from Startup file for AXM0F2*/
uint8_t _axm0f2_external_startup(void)
{
    uint8_t     start_cause = get_startcause();
    if (start_cause == STARTCAUSE_COLDSTART) {
    /* port configurations*/
    }
    if(start_cause == STARTCAUSE_COLDSTART)
    {
        coldstart = 1;          /* Cold start */
        return 0;               /* Variables init required */
    }
    else
    {
        coldstart = 0;          /* Warm start */
        return 1;               /* Variables init not required */
    }

}


void main (void)
{
       __enable_irq();

        #if defined USE_DBGLINK
            debuglink_init_axm0();              /// TODO
        #endif // USE_DBGLINK

    wtimer_init();
    if (coldstart)
    {
        led0_off();
        led1_off();
        led2_off();
```

```
        led3_off();
        axradio_setup_pincfg3();

        wakeup_desc.handler = wakeup_callback;
        i = axradio_init();
        axradio_set_local_address(&localaddr);
        axradio_set_default_remote_address(&remoteaddr);
       /* IMO calibration initialization and setup code start */
        /* Set Pin 0.0 (VTCXO pin) */
        GPIO_PRT0->DR_SET |= (1 << AXM0F2_VTCXO_PIN);

        /* IMO and ILO calibration setup */
        setup_osc_calibration(AXM0XX_HFCLK_CLOCK_FREQ, CLKSRC_RSYSCLK);
        /* IMO calibration initialization and setup code end */
       //configures receive mode
        i = axradio_set_mode(RADIO_MODE);


    }//coldtsart
    else
    {
      axradio_commsleepexit();
      NVIC_EnableIRQ(GPIOPort2_IRQn);
            /* ENABLE RIRQ PR5 2.4 INTERRUPT */
      GPIO_PRT2->INTR_CFG |= 1 << 8;
    }

    for (;;) {
        uint8_t flg;
        wtimer_runcallbacks();
        crit = enter_critical();


        flg = WTFLAG_CANSTANDBY;
        if (axradio_cansleep())
            flg |= WTFLAG_CANSLEEP;
        wtimer_idle(flg);
    }
}
```

8. FILES

This section lists the files that need to be included in an AXRadio project and what their purpose is.

File	Description
axradio.h	AXRadio API declaration
easyax5043.h	AXRadio AX5043 private header – AX5043 only
easyax5043.c	AXRadio AX5043 main code – AX5043 only
easyax5044_45.h	AXRadio AX5044_45 private header – for both AX5044 and AX5045
easyax5044_45.c	AXRadio AX5044_45 main code – for both AX5044 and AX5045
config.c	Parameters generated by AXRadioLab

9. Revision History

Version V3	Updated for SoC AXM0F243
Version V3	Updated Example Usage Code
Version V4	Added AX5044_45 references
Version V5	Removed redundant SoC references

10. Contact Information

ON Semiconductor
Oskar-Bider-Strasse 1
CH-8600 Dübendorf
SWITZERLAND

Phone +41 44 882 17 07
Fax +41 44 882 17 09
Email sales@onsemi.com
www.onsemi.com

For further product related or sales information please visit our website or contact your local representative.