

Software User Manual

LibMF (AX8052/AXM0 Support Library)



ON Semiconductor®

1. Introduction.....	9
2. Acronyms and Abbreviations.....	10
3. Microprocessor Functions.....	11
3.1. Header Files.....	11
3.1.1. ax8052.h, ax8052f131.h, ax8052f142.h, ax8052f143.h, ax8052f144_45.h, ax8052f151.h	11
3.1.2. axm0F2.h, axm0f243.h	11
3.2. ax8052regaddr.h	12
3.3. libmftypes.h.....	12
3.3.1. libmfversion	12
3.3.2. __code, __data, __xdata, __pdata, __generic	12
3.3.3. int8_t, int16_t, int32_t uint8_t, uint16_t, uint32_t.....	12
3.3.4. void delay(uint16_t us)	12
3.3.5. uint16_t random(void) uint16_t __data random_seed.....	12
3.3.6. uint8_t hweight8(uint8_t x) uint8_t hweight16(uint16_t x) uint8_t hweight32(uint32_t x).....	12
3.3.7. uint8_t parity8(uint8_t x) uint16_t parity16(uint16_t x) uint32_t parity32(uint32_t x) 13	
3.3.8. uint8_t rev8(uint8_t x)	13
3.3.9. int32_t signextend12(int16_t x) int32_t signextend16(int16_t x) int32_t signextend20(int32_t x) int32_t signextend24(int32_t x).....	13
3.3.10. int16_t signedlimit16(int16_t x, int16_t lim) int32_t signedlimit32(int32_t x, int32_t lim).....	13
3.3.11. uint8_t checksignedlimit16(int16_t x, int16_t lim) uint8_t checksignedlimit32(int32_t x, int32_t lim).....	13
3.3.12. void fmemset(void __generic *p, char c, uint16_t n)	13
3.3.13. void fmemcpy(void __generic *d, const void __generic *s, uint16_t n)	13
3.3.14. void wtimer_standby(void)	13
3.3.15. void enter_standby(void).....	14
3.3.16. void enter_sleep(void).....	14
3.3.17. void enter_sleep_cont(void).....	14
3.3.18. void enter_deepsleep(void)	14
3.3.19. WRNUM_SIGNED, WRNUM_PLUS, WRNUM_ZEROPLUS, WRNUM_PADZERO, WRNUM_TSDSEP, WRNUM_LCHEX	15
3.4. libmfdbglink.h	15
3.4.1. dbglink_init	16
3.4.1.1. AX8052 - void dbglink_init(void)	16

3.4.1.2.	AXM0F243 - void uart_init(uint8_t timernr, uint8_t wl, uint8_t stop).....	16
3.4.2.	uint8_t dbglink_rx(void).....	16
3.4.3.	void dbglink_tx(uint8_t v)	16
3.4.4.	void dbglink_writestr(const char *ch)	16
3.4.5.	void dbglink_writehex16(uint16_t val, uint8_t nrdig, uint8_t flags) void dbglink_writehex32(uint32_t val, uint8_t nrdig, uint8_t flags).....	16
3.4.6.	void dbglink_writenum16(uint16_t val, uint8_t nrdig, uint8_t flags) void dbglink_writenum32(uint32_t val, uint8_t nrdig, uint8_t flags)	16
3.4.7.	uint8_t dbglink_txbuffersize(void) uint8_t dbglink_rxbuffersize(void).....	17
3.4.8.	uint8_t dbglink_rxcount(void).....	17
3.4.9.	uint8_t dbglink_txfree(void).....	17
3.4.10.	uint8_t dbglink_txidle(void)	17
3.4.11.	void dbglink_wait_rxcount(uint8_t v)	17
3.4.12.	void dbglink_wait_txdone(void).....	17
3.4.13.	void dbglink_wait_txfree(uint8_t v)	17
3.4.14.	void dbglink_rxadvance(uint8_t idx).....	17
3.4.15.	void dbglink_txadvance(uint8_t idx)	17
3.4.16.	uint8_t dbglink_rxpeek(uint8_t idx)	18
3.4.17.	void dbglink_txpoke(uint8_t idx, uint8_t ch)	18
3.4.18.	void dbglink_txpokehex(uint8_t idx, uint8_t ch).....	18
3.4.19.	uint8_t dbglink_poll(void)	18
3.4.20.	DBGLINK_DEFINE_TXBUFFER(sz) DBGLINK_DEFINE_RXBUFFER(sz) 18	
3.5.	libmfuart.h.....	18
3.5.1.	AX8052.....	18
	void uart_timer0_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	18
	void uart_timer1_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	18
	void uart_timer2_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	18
3.5.2.	AXM0F243	19
	void uart_timer0_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
	void uart_timer1_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
	void uart_timer2_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
	void uart_timer3_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
	void uart_timer4_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
	void uart_timer5_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
	void uart_timer6_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19

void uart_timer7_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
void uart_timer8_baud(uint8_t clksrc, uint32_t baud, uint32_t clkfreq).....	19
3.6. libmfuart0.h, libmfuart1.h	19
3.6.1. void uart0_init(uint8_t timernr, uint8_t wl, uint8_t stop) void uart0_init(uint8_t timernr, uint8_t wl, uint8_t stop)	19
3.6.2. void uart0_stop(void) void uart1_stop(void)	20
3.6.3. uint8_t uart0_rx(void) uint8_t uart1_rx(void)	20
3.6.4. void uart0_tx(uint8_t v) void uart1_tx(uint8_t v).....	20
3.6.5. void uart0_writestr(const char *ch) void uart1_writestr(const char *ch)	20
3.6.6. void uart0_writehex16(uint16_t val, uint8_t nrdig, uint8_t flags) void uart0_writehex32(uint32_t val, uint8_t nrdig, uint8_t flags) void uart1_writehex16(uint16_t val, uint8_t nrdig, uint8_t flags) void uart1_writehex32(uint32_t val, uint8_t nrdig, uint8_t flags) 20	
3.6.7. void uart0_writenum16(uint16_t val, uint8_t nrdig, uint8_t flags) void uart0_writenum32(uint32_t val, uint8_t nrdig, uint8_t flags) void uart1_writenum16(uint16_t val, uint8_t nrdig, uint8_t flags) void uart1_writenum32(uint32_t val, uint8_t nrdig, uint8_t flags)	20
3.6.8. uint8_t uart0_txbufferize(void) uint8_t uart1_txbufferize(void) uint8_t uart0_rxbufferize(void) uint8_t uart1_rxbufferize(void).....	21
3.6.9. uint8_t uart0_rxcount(void) uint8_t uart1_rxcount(void)	21
3.6.10. uint8_t uart0_txfree(void) uint8_t uart1_txfree(void).....	21
3.6.11. uint8_t uart0_txidle(void) uint8_t uart1_txidle(void)	21
3.6.12. void uart0_wait_rxcount(uint8_t v) void uart1_wait_rxcount(uint8_t v)	21
3.6.13. void uart0_wait_txdone(void) void uart1_wait_txdone(void)	21
3.6.14. void uart0_wait_txfree(uint8_t v) void uart1_wait_txfree(uint8_t v).....	22
3.6.15. void uart0_rxadvance(uint8_t idx) void uart1_rxadvance(uint8_t idx)	22
3.6.16. void uart0_txadvance(uint8_t idx) void uart1_txadvance(uint8_t idx).....	22
3.6.17. uint8_t uart0_rxpeek(uint8_t idx) uint8_t uart1_rxpeek(uint8_t idx).....	22
3.6.18. void uart0_txpoke(uint8_t idx, uint8_t ch) void uart1_txpoke(uint8_t idx, uint8_t ch)22	
3.6.19. void uart0_txpokehex(uint8_t idx, uint8_t ch) void uart1_txpokehex(uint8_t idx, uint8_t ch)22	
3.6.20. uint8_t uart0_poll(void) uint8_t uart1_poll(void).....	22
3.6.21. UART0_DEFINE_TXBUFFER(sz) UART1_DEFINE_TXBUFFER(sz) UART0_DEFINE_RXBUFFER(sz) UART1_DEFINE_RXBUFFER(sz)	23
3.7. libmfflash.h	23
3.7.1. void flash_unlock(void)	23
3.7.2. void flash_lock(void)	23

3.7.3.	int8_t flash_pageerase(uint16_t pgaddr).....	23
3.7.4.	int8_t flash_write(uint16_t waddr, uint16_t wdata)	23
3.7.5.	uint16_t flash_read(uint16_t raddr)	24
3.7.6.	uint8_t flash_apply_calibration(void).....	24
3.8.	libmfradio.h.....	24
3.8.1.	uint8_t radio_read8(uint16_t addr)	24
	uint16_t radio_read16(uint16_t addr) uint32_t radio_read24(uint16_t addr) uint32_t radio_read32(uint16_t addr)	24
3.8.2.	void radio_write8(uint16_t addr, uint8_t d).....	24
	void radio_write16(uint16_t addr, uint16_t d) void radio_write24(uint16_t addr, uint32_t d) void radio_write32(uint16_t addr, uint32_t d)	24
3.8.3.	uint8_t ax5031_reset(void) uint8_t ax5042_reset(void) uint8_t ax5043_reset(void)	25
	uint8_t ax5044_45_reset(void) uint8_t ax5051_reset(void).....	25
3.8.4.	void ax5031_commsleepexit(void) void ax5042_commsleepexit(void) void ax5043_commsleepexit(void).....	25
	void ax5044_45_commsleepexit(void) void ax5051_commsleepexit(void).....	25
3.8.5.	void ax5031_comminit(void) void ax5042_comminit(void) void ax5043_comminit(void).....	25
	void ax5044_45_comminit(void) void ax5051_comminit(void).....	25
3.8.6.	void ax5031_rclk_enable(uint8_t div) void ax5042_rclk_enable(uint8_t div) void ax5043_rclk_enable(uint8_t div)	25
	void ax5044_45_rclk_enable(uint8_t div) void ax5051_rclk_enable(uint8_t div)	25
3.8.7.	void ax5042_rclk_disable(void) void ax5031_rclk_disable(void) void ax5043_rclk_disable(void)	26
	void ax5044_45_rclk_disable(void) void ax5051_rclk_disable(void)	26
3.8.8.	void ax5043_rclk_wait_stable(uint8_t wtfld)	26
	void ax5044_45_rclk_wait_stable(uint8_t wtfld)	26
3.8.9.	void ax5043_enter_deepsleep(void)	26
	void ax5044_45_enter_deepsleep(void)	26
3.8.10.	uint8_t ax5043_wakeup_deepsleep(void).....	26
	uint8_t ax5044_45_wakeup_deepsleep(void)	26
3.8.11.	void ax5043_readfifo(uint8_t *ptr, uint8_t len).....	26
	void ax5044_45_readfifo(uint8_t *ptr, uint8_t len)	26
3.8.12.	void ax5043_writefifo(const uint8_t *ptr, uint8_t len)	26
	void ax5044_45_writefifo(const uint8_t *ptr, uint8_t len).....	26
3.8.13.	void ax5043_set_pwramp_pin(uint8_t config_pin)	26

void ax5044_45_set_pwramp_pin(uint8_t config_pin).....	26
3.8.14. uint8_t ax5043_get_pwramp_pin()	27
uint8_t ax5044_45_get_pwramp_pin().....	27
3.8.15. void ax5043_set_antsel_pin(uint8_t config_pin).....	27
void ax5044_45_set_antsel_pin(uint8_t config_pin)	27
3.8.16. uint8_t ax5043_get_antsel_pin().....	27
uint8_t ax5044_45_get_antsel_pin()	27
3.9. libmfadc.h.....	27
3.9.1. int16_t adc_measure_temperature(void)	27
3.9.2. uint16_t adc_singleended_offset_x1(void).....	28
3.10. libmfwtimer.h	28
3.10.1. uint32_t wtimer0_curtime(void) uint32_t wtimer1_curtime(void).....	30
3.10.2. void wtimer0_addabsolute(__xdata struct wtimer_desc *desc) void wtimer1_addabsolute(__xdata struct wtimer_desc *desc) void wtimer0_addrelative(__xdata struct wtimer_desc *desc) void wtimer1_addrelative(__xdata struct wtimer_desc *desc).....	30
3.10.3. uint8_t wtimer0_remove(__xdata struct wtimer_desc *desc) uint8_t wtimer1_remove(__xdata struct wtimer_desc *desc) uint8_t wtimer_remove(__xdata struct wtimer_desc *desc).....	30
3.10.4. void wtimer_add_callback(__xdata struct wtimer_callback *desc)	30
3.10.5. uint8_t wtimer_remove_callback(__xdata struct wtimer_callback *desc)	30
3.10.6. uint8_t wtimer_idle(uint8_t flags)	31
3.10.7. uint8_t wtimer_runcallbacks(void)	31
3.10.8. uint8_t wtimer_cansleep(void).....	31
3.10.9. void wtimer0_setclksrc(uint8_t clksrc, unit8_t prescaler) void wtimer1_setclksrc(uint8_t clksrc, unit8_t prescaler)	31
3.10.10. void wtimer_init(void)	32
3.10.11. void wtimer_init_deepsleep(void).....	32
3.11. libmfosc.h	32
3.11.1. void turn_off_xosc(void).....	32
3.11.2. void turn_off_lpxosc(void)	32
3.11.3. void setup_xosc(void)	32
3.11.4. void setup_lpxosc(void)	32
3.11.5. uint8_t setup_osc_calibration(uint32_t reffreq, uint8_t refosc).....	32
3.11.6. setup_osc_calibration_const(reffreq,refosc)	33
3.11.7. uint32_t wtimer0_correctinterval(uint32_t intvl).....	33
uint32_t wtimer1_correctinterval(uint32_t intvl)	33

3.12.	Critical section.....	33
3.12.1.	criticalsection_t enter_critical(void)	33
3.12.2.	void exit_critical(criticalsection_t crit)	34
3.12.3.	void reenter_critical(void).....	34
3.13.	SPI Driver	34
3.13.1.	uint8_t spi0_init(uint8_t speed, uint8_t sel_div, uint8_t mode, uint8_t bitorder) 34	
3.13.2.	void spi0_read(uint8_t *rdbuf, uint8_t rdlength)	35
3.13.3.	void spi0_write(uint8_t *wrbuf, uint8_t wrlength).....	35
3.13.4.	uint8_t spi0_wait_until_done(void)	35
3.13.5.	uint8_t spi0_get_status(void)	35
3.13.6.	uint8_t spi0_close(void).....	35
3.14.	I2C Driver.....	37
3.14.1.	uint8_t i2c0_init(uint8_t mode, uint8_t speed, uint8_t sel_div, uint8_t clk_src)38 uint8_t i2c1_init(uint8_t mode, uint8_t speed, uint8_t sel_div, uint8_t clk_src)	38
3.14.2.	void i2c0_read(uint16_t slaveAddress, uint8_t *rdData, uint8_t length, uint8 repeated) 38 void i2c1_read(uint16_t slaveAddress, uint8_t *rdData, uint8_t length, uint8 repeated)	38
3.14.3.	void i2c0_write(uint16_t slaveAddress, uint8_t *wrData, uint8_t length, uint8 repeated) 39 void i2c1_write(uint16_t slaveAddress, uint8_t *wrData, uint8_t length, uint8 repeated) ..	39
3.14.4.	uint8_t i2c0_wait_until_done(void)	39
	uint8_t i2c1_wait_until_done(void).....	39
3.14.5.	uint8_t i2c0_get_status(void).....	39
	uint8_t i2c1_get_status(void).....	39
3.14.6.	uint8_t i2c0_close(void).....	40
	uint8_t i2c1_close(void)	40
3.15.	ADC Driver	42
3.15.1.	uint8_t adc_init(uint8_t positive_input, uint8_t negative_input, uint8_t conv_mode, uint8_t ref_input, uint16_t ext_vref).....	42
3.15.2.	void adc_start_measurement (void)	43
3.15.3.	uint8_t adc_wait_until_done (void)	43
3.15.4.	uint8_t adc_getstatus (void)	43
3.15.5.	uint8_t adc_get_result (void)	43
3.15.6.	int16_t adc_result_in_mVolts(int16_t adcCounts).....	44
3.15.7.	int32_t adc_convert_dietemp(int32_t adcCounts)	44
3.15.8.	void adc_close(void)	44

3.16.	PWM Driver	48
3.16.1.	uint8_t pwm_init(uint8_t sel_pwm, uint8_t pwm_pin, uint8_t pwm_type, uint32_t pwm_period, uint8_t pwm_dutycycle, uint8_t sel_div)	48
3.16.2.	uint8_t pwm_dutycycle (uint8_t sel_pwm, uint8_t pwm_dutycycle)	49
3.16.3.	uint8_t pwm_start (uint8_t sel_pwm)	49
3.16.4.	uint8_t pwm_stop (uint8_t sel_pwm)	50
3.16.5.	uint8_t pwm_close (uint8_t sel_pwm)	50
4.	Convenience Functions	53
4.1.	libmfbch.h	53
4.1.1.	uint16_t bch3121_syndrome(uint32_t cw)	53
4.1.2.	uint32_t bch3121_encode(uint32_t cw)	53
4.1.3.	uint32_t bch3121_encode_parity(uint32_t cw)	53
4.1.4.	uint32_t bch3121_decode(uint32_t cw)	53
4.1.5.	uint32_t bch3121_decode_parity(uint32_t cw)	53
4.2.	libmfcrh.h	53
4.2.1.	uint8_t crc8_ccitt_byte(uint8_t crc, uint8_t c)	53
4.2.2.	uint8_t crc8_ccitt(const uint8_t *buf, uint8_t len, uint8_t init)	54
4.2.3.	uint8_t crc8_onewire_byte(uint8_t crc, uint8_t c)	54
4.2.4.	uint8_t crc8_onewire(const uint8_t *buf, uint8_t len, uint8_t init)	54
4.2.5.	uint8_t crc_crc8ccitt_byte(uint8_t crc, uint8_t c)	54
4.2.6.	uint8_t crc_crc8ccitt_msb_byte(uint8_t crc, uint8_t c)	54
4.2.7.	uint8_t crc_crc8onewire_byte(uint8_t crc, uint8_t c)	54
4.2.8.	uint8_t crc_crc8onewire_msb_byte(uint8_t crc, uint8_t c)	54
4.2.9.	uint16_t crc_ccitt_byte(uint16_t crc, uint8_t c)	55
4.2.10.	uint16_t crc_ccitt_msb_byte(uint16_t crc, uint8_t c)	55
4.2.11.	uint16_t crc_crc16_byte(uint16_t crc, uint8_t c)	55
4.2.12.	uint16_t crc_crc16_msb_byte(uint16_t crc, uint8_t c)	55
4.2.13.	uint16_t crc_crc16dnp_byte(uint16_t crc, uint8_t c)	55
4.2.14.	uint16_t crc_crc16dnp_msb_byte(uint16_t crc, uint8_t c)	55
4.2.15.	uint32_t crc_crc32_byte(uint32_t crc, uint8_t c)	56
4.2.16.	uint32_t crc_crc32_msb_byte(uint32_t crc, uint8_t c)	56
4.2.17.	uint8_t crc_crc8ccitt(const uint8_t *buf, uint16_t buflen, uint8_t crc) uint8_t crc_crc8ccitt_msb(const uint8_t *buf, uint16_t buflen, uint8_t crc) uint8_t crc_crc8onewire(const uint8_t *buf, uint16_t buflen, uint8_t crc) uint8_t crc_crc8onewire_msb(const uint8_t *buf, uint16_t buflen, uint8_t crc) uint16_t crc_ccitt(const uint8_t *buf, uint16_t buflen, uint16_t crc) uint16_t crc_ccitt_msb(const uint8_t *buf, uint16_t buflen, uint16_t crc) uint16_t crc_crc16(const uint8_t *buf, uint16_t	

buflen, uint16_t crc) uint16_t crc_crc16_msb(const uint8_t *buf, uint16_t buflen, uint16_t crc) uint16_t crc_crc16dnp(const uint8_t *buf, uint16_t buflen, uint16_t crc) uint16_t crc_crc16dnp_msb(const uint8_t *buf, uint16_t buflen, uint16_t crc) uint32_t crc_crc32(const uint8_t *buf, uint16_t buflen, uint32_t crc) uint32_t crc_crc32_msb(const uint8_t *buf, uint16_t buflen, uint32_t crc).....	56
4.2.18. uint16_t pn9_advance(uint16_t pn9)	57
4.2.19. uint16_t pn9_advance_bit(uint16_t pn9)	57
4.2.20. uint16_t pn9_advance_bits(uint16_t pn9, uint16_t bits)	57
4.2.21. uint16_t pn9_advance_byte(uint16_t pn9).....	57
4.2.22. uint16_t pn9_buffer(uint8_t __generic *buf, uint16_t buflen, uint16_t pn9, uint8_t xor).....	57
5. HISTORY	58
6. Contact Information	59

1. INTRODUCTION

LibMF is a convenience library to ease the use of AX8052/AXM0 Microprocessor and Evaluation Board features. It contains the following features:

- DebugLink UART
- RS-232 UART
- FLASH writing under Software Control
- Radio Initialization and Probing
- CRC-8 and CRC-16 routines
- Evaluation Board LCD access

LibMF is available in source and binary form for SDCC, Keil C51 and IAR ICC for AX8052 MCU.

LibMF is available in source and binary form for ARM-GCC for AXM0 MCU.

Unless explicitly mentioned otherwise, details outlined in this document are applicable to both AX8052 and AXM0.

2. ACRONYMS AND ABBREVIATIONS

AX8052	MCU 8052
AX8052F143	MCU 8052 + RADIO AX5043
AX8052F144_45	MCU 8052 + RADIO AX5044/45
AXM0	MCU ARM Cortex M0 Plus
AXM0F243	MCU ARM Cortex M0 Plus + RADIO AX5043
SDCC	Small Device C Compiler
GCC	GNU Compiler Collection
WCOWDT	Watch Crystal Oscillator Watch Dog Timer
TCPWM	Timer, Counter, and Pulse Width Modulator Block

3. MICROPROCESSOR FUNCTIONS

3.1. HEADER FILES

3.1.1. AX8052.H, AX8052F131.H, AX8052F142.H, AX8052F143.H, AX8052F144_45.H, AX8052F151.H

These headers define the special function registers (SFR) of the ON Semiconductor AX8052 Microprocessor. `ax8052.h` defines the SFR's of the microcontroller core only. It is suitable for the AX8052F101, or the AX8052F100 without any ON Semiconductor radio chip connected. `ax8052f131.h`, `ax8052f142.h`, `ax8052f143.h`, `ax8052f144_45.h` and `ax8052f151.h` define the special function registers of the microcontroller core as well as the respective radio peripheral. So for example `ax8052f151.h` is suitable for the SoC AX8052F151, as well as the two-chip combination AX8052F100 plus AX5051.

3.1.2. AXM0F2.H, AXM0F243.H

These headers define the registers of the AXM0 Microprocessor. `axm0f2.h` defines the registers of the microcontroller core only. It is suitable for the AXM0 without any radio chip connected. `axm0f243.h` defines the registers of the microcontroller core as well as the respective radio peripheral. So for example `axm0f243.h` is suitable for the SoC AXM0F243, as well as the two-chip combination AXM0F2 plus AX5043.

3.2. AX8052REGADDR.H

This header provides defines for the AX8052 special function register (SFR) addresses. Contrary to ax8052.h, which provides defines that, when used like variables, access the registers, the ax8052regaddr.h header file only provides defines for the register addresses.

3.3. LIBMFTYPES.H

3.3.1. LIBMFVERSION

This preprocessor define specifies the version of the libmf library. It can be used for version specific processing, for example to accommodate incompatible changes to libmf.

3.3.2. __CODE, __DATA, __XDATA, __PDATA, __GENERIC

These defines provide compiler independent address space specifiers.

3.3.3. INT8_T, INT16_T, INT32_T UINT8_T, UINT16_T, UINT32_T

These are C99 style sized integer types. u signifies unsigned (the default is signed), and the number specifies the size in bits. So uint16_t is a 16 bit (2 byte) unsigned integer.

3.3.4. VOID DELAY(UINT16_T US)

This routine busy waits for the given number of microseconds. The wait duration is approximately correct when the microprocessor runs at 20MHz for AX8052 and 48MHz for AXM0F243.

This routine should not be used for long delays for power consumption reasons. For long delays, enter standby or sleep mode and wake up using the wakeup timer peripheral.

3.3.5. UINT16_T RANDOM(VOID) UINT16_T __DATA RANDOM_SEED

This function provides a simple linear congruential random number generator. Its seed is stored in the variable random_seed. This is not a cryptographic quality random number generator. For cryptographic quality random number, use the true random number generator peripheral and mix its output sufficiently.

3.3.6. UINT8_T HWEIGHT8(UINT8_T X) UINT8_T HWEIGHT16(UINT16_T X) UINT8_T HWEIGHT32(UINT32_T X)

These functions compute the hamming weight or population count, i.e. the number of bits set.

```

3.3.7.      UINT8_T PARITY8(UINT8_T X)
            UINT16_T PARITY16(UINT16_T X)
            UINT32_T PARITY32(UINT32_T X)

```

These functions compute the odd parity of the argument, i.e. they return one if the argument contains an odd number of bits set.

```

3.3.8.      UINT8_T REV8(UINT8_T X)

```

This functions reverses the bits of its argument, i.e. Bits 0 and 7 are swapped, Bits 1 and 6, and so on.

```

3.3.9.      INT32_T SIGNEXTEND12(INT16_T X)
            INT32_T SIGNEXTEND16(INT16_T X)
            INT32_T SIGNEXTEND20(INT32_T X)
            INT32_T SIGNEXTEND24(INT32_T X)

```

These functions sign extend a given number.

```

3.3.10.     INT16_T SIGNEDLIMIT16(INT16_T X, INT16_T LIM)
            INT32_T SIGNEDLIMIT32(INT32_T X, INT32_T LIM)

```

This function returns $-lim$ if x is less than $-lim$, lim if x is more than lim , and x otherwise. lim must be in the range from $0...2^{15}-1$ or $0...2^{31}-1$ for these functions to work correctly.

```

3.3.11.     UINT8_T CHECKSIGNEDLIMIT16(INT16_T X, INT16_T LIM)
            UINT8_T CHECKSIGNEDLIMIT32(INT32_T X, INT32_T LIM)

```

These functions return zero if x is less than $-lim$ or more than lim , and one otherwise. lim must be in the range from $0...2^{15}-1$ or $0...2^{31}-1$ for these functions to work correctly.

```

3.3.12.     VOID FMEMSET(VOID __GENERIC *P, CHAR C, UINT16_T N)

```

`fmemset` is a fast version of the standard C library `memset` function. It however takes a generic pointer, and internally uses separate implementations depending on the address space the pointer points to.

```

3.3.13.     VOID FMEMCPY(VOID __GENERIC *D, CONST VOID __GENERIC *S,
                        UINT16_T N)

```

`memcpy` is a fast version of the standard C library `memcpy` function. It however takes generic pointers, and internally uses separate implementations depending on the address space combinations the pointers point to.

```

3.3.14.     VOID WTIMER_STANDBY(VOID)

```

On SDCC, this function calls `wtimer_runcallbacks` and `wtimer_idle(WTFLAG_CANSTANDBY)` if the wakeup timer module is used in the project. Otherwise, or when using Keil IAR or ARM-GCC it calls `enter_standby()`, below.

3.3.15. VOID ENTER_STANDBY(VOID)

AX8052:

This macro enters standby mode. `enter_standby` should be preferred over directly writing to `PCON`, as it includes a NOP instruction after writing to `PCON`, to ensure correct processing.

AXM0F243:

This macro takes chip into sleep mode.

3.3.16. VOID ENTER_SLEEP(VOID)

AX8052:

This routine enters the sleep mode. `enter_sleep` should be preferred over directly writing to `PCON`, as it ensures the same behaviour whether or not the debugger is attached.

An incompatible change to this macro has been made. Earlier libmf versions took an argument to specify which XRAM blocks should be kept. This argument has been removed. The XRAM blocks to keep should be directly written to `PCON` (with bits 1:0 written to 00). This change has been done for efficiency.

After wake-up, program execution restarts at the reset vector. Reset and wake-up can be distinguished by `PCON.6`.

AXM0F243:

This routine takes chip into deep sleep mode. Chip will not enter deep sleep mode if the debugger is enabled.

After wake-up, program execution restarts at the `Reset_Handler_NoStartCause`, defined in startup file.

3.3.17. VOID ENTER_SLEEP_CONT(VOID)

AX8052:

This routine enters the sleep mode. Contrary to `enter_sleep()`, this function returns after wake-up and execution continues. `_sdcc_external_startup()` is called after wake-up, before the function returns. All peripherals, except the system controller, need to be reinitialized (but note silicon revision V1 errata).

This function is only available for SDCC. It is incompatible with the external stack.

AXM0F243:

This routine enters the sleep mode. Contrary to `enter_sleep()`, this function returns after wake-up and execution continues.

3.3.18. VOID ENTER_DEEPSLEEP(VOID)

AX8052:

This macro causes the processor to enter deepsleep mode. It should be preferred over directly writing to PCON, as it ensures the same behaviour whether or not the debugger is attached.

AXM0F243:

This macro is not available.

3.3.19. WRNUM_SIGNED, WRNUM_PLUS, WRNUM_ZEROPLUS, WRNUM_PADZERO, WRNUM_TSDSEP, WRNUM_LCHEX

These or'able constants are used as flags to the wrnum16/wrnum32/wrhex16/wrhex32 family of functions to specify the behavior.

- ☐ WRNUM_SIGNED: treat the argument as signed number. The default is unsigned.
- ☐ WRNUM_PLUS: Output a plus sign for positive number. Default is to output a sign character only for negative numbers.
- ☐ WRNUM_ZEROPLUS: Output a plus sign for zero arguments. Default is to omit a sign character for zeros.
- ☐ WRNUM_PADZERO: Pad the output with leading zeros. Default is to pad the output with space characters.
- ☐ WRNUM_TSDSEP: Output a thousands separator (apostrophe). Default is to not output a thousands separator. For hexadecimal output, the separator separates word (16bit) numbers.
- ☐ WRNUM_LCHEX: Output upper case hexadecimal characters. Default is to output lowercase hexadecimal characters.

3.4. LIBMFDGLINK.H

AX8052:

DebugLink is an UART whose input/output is sent over the debug link interface to the debugger. It is displayed in a window of the AxCode::Blocks IDE.

AXM0F243:

DebugLink is an UART whose input/output is sent over the debug link interface to the debugger. DebugLink output can be displayed on a PC serial terminal tool.

There are two UART's available for the user, UART 0 and UART 1. Select UART that needs to be used for debug Link using macro DBGLINK_UART(x), where x can be either 0 or 1. The selected uart will be bound to debug link.

When using SDCC/ARM-GCC, libmfdbglink.h must be included in the file containing the definition for main.

3.4.1. DBGLINK_INIT

3.4.1.1. AX8052 - VOID DBGLINK_INIT(VOID)

dbglink_init initializes the DebugLink driver. It installs an interrupt handler and initializes the FIFOs (64 Bytes in XRAM (AX8052) or RAM (AXM0) for each direction). It must be called before using DebugLink, and should be called before interrupts are globally enabled.

3.4.1.2. AXM0F243 - VOID UART_INIT(UINT8_T TIMERNR, UINT8_T WL, UINT8_T STOP)

dbglink_init initializes the DebugLink driver. It installs an interrupt handler and initializes the FIFOs in RAM for each direction. It must be called before using DebugLink, and should be called before interrupts are globally enabled.

timernr (0 to 5) specifies the timer that should be used as baud rate generator. The corresponding timer must be initialized with uart_timerx_baud.

wl (5...9) specifies the word length, i.e. the number of bits between start and stop bit(s).

stop (1...2) specifies the number of stop bits. This affects only the transmitter; the receiver always accepts 1 stop bit.

3.4.2. UINT8_T DBGLINK_RX(VOID)

dbglink_rx reads a character from the receive FIFO. It blocks until a character is available. If blocking is undesirable, it should only be called when dbglink_rxcount() returns a nonzero result.

3.4.3. VOID DBGLINK_TX(UINT8_T V)

dbglink_tx puts one character into the transmit FIFO and starts the transmitter, if it is not already running. It blocks if the FIFO is full. If blocking is not desirable, it should only be called if dbglink_txfree() returns a nonzero value.

3.4.4. VOID DBGLINK_WRITESTR(CONST CHAR *CH)

dbglink_writestr writes the null terminated C string to the DebugLink interface. It is blocking if the string exceeds the FIFO free space.

3.4.5. VOID DBGLINK_WRITEHEX16(UINT16_T VAL, UINT8_T NRDIG, UINT8_T FLAGS) VOID DBGLINK_WRITEHEX32(UINT32_T VAL, UINT8_T NRDIG, UINT8_T FLAGS)

dbglink_writehex16 and dbglink_writehex32 write a hexadecimal number to the DebugLink interface. The number of desired digits must be given in the nrdig parameter. It is blocking if the number of digits exceed the FIFO free space. flags is a bitwise or combination of the WRNUM constants documented in section 3.3.19.

3.4.6. VOID DBGLINK_WRITENUM16(UINT16_T VAL, UINT8_T NRDIG, UINT8_T FLAGS)


```
VOID DBGLINK_WRITENUM32(UINT32_T VAL, UINT8_T NRDIG, UINT8_T  
FLAGS)
```

dbglink_writenum16 and dbglink_writenum32 write a hexadecimal number to the DebugLink interface. The number of desired digits must be given in the nrdig parameter. It is blocking if the number of digits exceed the FIFO free space. flags is a bitwise or combination of the WRNUM constants documented in section 3.3.19.

```
3.4.7.      UINT8_T DBGLINK_TXBUFFERSIZE(VOID)  
            UINT8_T DBGLINK_RXBUFFERSIZE(VOID)
```

These functions return the transmit and receive buffer sizes. Buffer sizes are configurable.

```
3.4.8.      UINT8_T DBGLINK_RXCOUNT(VOID)
```

dbglink_rxcount returns the number of characters in the receive FIFO.

```
3.4.9.      UINT8_T DBGLINK_TXFREE(VOID)
```

dbglink_txfree returns the number of free space for characters in the transmit FIFO.

```
3.4.10.     UINT8_T DBGLINK_TXIDLE(VOID)
```

dbglink_txidle returns one if the transmitter is idle, i.e. all transmit characters have left the wire. This routine can be used to determine whether the microprocessor can enter sleep.

```
3.4.11.     VOID DBGLINK_WAIT_RXCOUNT(UINT8_T V)
```

dbglink_wait_rxcount blocks until the number of characters in the receive FIFO reach or exceed v.

```
3.4.12.     VOID DBGLINK_WAIT_TXDONE(VOID)
```

dbglink_wait_txdone blocks until the last character in the transmit FIFO has been transmitted.

```
3.4.13.     VOID DBGLINK_WAIT_TXFREE(UINT8_T V)
```

dbglink_wait_txfree blocks until the number of free space for characters in the transmit FIFO reaches or exceeds v.

```
3.4.14.     VOID DBGLINK_RXADVANCE(UINT8_T IDX)
```

dbglink_rxadvance drops the given number of characters from the front of the FIFO.

```
3.4.15.     VOID DBGLINK_TXADVANCE(UINT8_T IDX)
```

dbglink_txadvance adds the given number of characters at the end of the transmit FIFO. They must have been defined by dbglink_txpoke or dbglink_txpokehex before, otherwise the transmitted characters are undefined.

3.4.16. `UINT8_T DBGLINK_RXPEEK(UINT8_T IDX)`

`dbglink_rxpeek` returns the `idx`'th character in the receive FIFO without modifying the FIFO.

3.4.17. `VOID DBGLINK_TXPOKE(UINT8_T IDX, UINT8_T CH)`

`dbglink_txpoke` puts a character at the `idx`'th position after the end of the transmit FIFO. It does not become part of the FIFO. In order to actually transmit the character, `dbglink_txadvance` must be called.

3.4.18. `VOID DBGLINK_TXPOKEHEX(UINT8_T IDX, UINT8_T CH)`

`dbglink_txpoke` puts a hexadecimal character at the `idx`'th position after the end of the transmit FIFO. It does not become part of the FIFO. In order to actually transmit the character, `dbglink_txadvance` must be called.

3.4.19. `UINT8_T DBGLINK_POLL(VOID)`

Normally, data is transferred between the FIFO and the hardware using an interrupt handler. Sometimes though, an interrupt handler is undesirable; in this case, `dbglink_poll` can be called periodically to transfer data between the FIFO and the hardware if available. It returns a bit mask with bit 0 set if a byte was transferred from the hardware to the receive FIFO, and bit 1 set if a byte was transferred from the transmit FIFO to the hardware.

3.4.20. `DBGLINK_DEFINE_TXBUFFER(SZ)` `DBGLINK_DEFINE_RXBUFFER(SZ)`

These macros defines transmit and receive buffer sizes, respectively. The argument needs to lie between 2 and 256 (inclusive). Note that the argument specifies the total buffer size; one character is unusable due to the design of the buffer pointers, therefore `dbglink_txbuffer_size` and `dbglink_rxbuffer_size` will return one byte less than the argument.

3.5. LIBMFUART.H

This header file defines routines that are common to both UARTs.

AXM0F243:

There are two UARTs available in AXM0F243, UART 0 and 1. The UART that is not bound to DebugLink is available to use with in this module.

3.5.1. AX8052

```
VOID UART_TIMER0_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER1_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER2_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
```

Each UART needs to be paired with a general purpose Timer for Baud rate generation. Both UARTs can be paired with the same Timer if the same Baud rate is desired. These routines set up a particular timer for baud rate generation.

clksrc may be one of CLKSRC_FRCOSC, CLKSRC_LPOSC, CLKSRC_XOSC, CLKSRC_LPXOSC, CLKSRC_RSYSCLK, CLKSRC_TCLK, CLKSRC_SYSCLK or CLKSRC_OFF.

If clksrc is CLKSRC_XOSC or CLKSRC_LPXOSC, the frequency of the connected crystal must be given as argument xtalclk (in Hz).

Baud is the baudrate in bits/s, while clkfreq is the clock frequency of the selected oscillator.

3.5.2. AXM0F243

```
VOID UART_TIMER0_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER1_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER2_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER3_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER4_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER5_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER6_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER7_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
VOID UART_TIMER8_BAUD(UINT8_T CLKSRC, UINT32_T BAUD, UINT32_T CLKFREQ)
```

Each UART needs to be paired with a general purpose Timer for Baud rate generation. Both UARTs can be paired with the same Timer if the same Baud rate is desired. These routines set up a particular timer for baud rate generation.

Clksrc: By default clksrc is HSOSC and Clksrc parameter is ignored.

Baud is the baudrate in bits/s. Eg: 9600, 14400, 19200, 28800, 38400, 56000, 57600, 11520 etc.

clkfreq is the clock frequency of the HSOSC oscillator.

3.6. LIBMFUART0.H, LIBMFUART1.H

When using SDCC/ARM-GCC, libmfuart0.h/libmfuart1.h must be included in the file containing the definition for main.

```
3.6.1. VOID UART0_INIT(UINT8_T TIMERNR, UINT8_T WL, UINT8_T STOP)
VOID UART0_INIT(UINT8_T TIMERNR, UINT8_T WL, UINT8_T STOP)
```

uart0_init / uart1_init initializes the UART driver. It installs an interrupt handler and initializes the FIFOs (64 Bytes in XRAM for AX8052 or in RAM for AXM0 for each direction and UART). It must be called before using the UART, and should be called before interrupts are globally enabled.

timernr (0 to 2 for AX8052 or 0 to 5 for AXM0F243) specifies the timer that should be used as baud rate generator. The corresponding timer must be initialized with uart_timerx_baud.

wl (5...9) specifies the word length, i.e. the number of bits between start and stop bit(s).

stop (1...2) specifies the number of stop bits. This affects only the transmitter; the receiver always accepts 1 stop bit.

3.6.2. VOID UART0_STOP(VOID)
 VOID UART1_STOP(VOID)

uart0_stop / uart1_stop can be used to stop the UART after it has been initialized with uart0_init / uart1_init. After stopping, no reads or writes to the UART should be performed. In order to restart the UART, uart0_init / uart1_init must be used.

3.6.3. UINT8_T UART0_RX(VOID)
 UINT8_T UART1_RX(VOID)

uart0_rx / uart1_rx reads a character from the receive FIFO. It blocks until a character is available. If blocking is undesirable, it should only be called when uart0_rxcount() / uart1_rxcount() returns a nonzero result.

3.6.4. VOID UART0_TX(UINT8_T V)
 VOID UART1_TX(UINT8_T V)

uart0_tx / uart1_tx puts one character into the transmit FIFO and starts the transmitter, if it is not already running. It blocks if the FIFO is full. If blocking is not desirable, it should only be called if uart0_txfree() / uart1_txfree() returns a nonzero value.

3.6.5. VOID UART0_WRITESTR(CONST CHAR *CH)
 VOID UART1_WRITESTR(CONST CHAR *CH)

uart0_writestr / uart1_writestr writes the null terminated C string to the UART. It is blocking if the string exceeds the FIFO free space.

3.6.6. VOID UART0_WRITEHEX16(UINT16_T VAL, UINT8_T NRDIG, UINT8_T
 FLAGS)
 VOID UART0_WRITEHEX32(UINT32_T VAL, UINT8_T NRDIG, UINT8_T
 FLAGS)
 VOID UART1_WRITEHEX16(UINT16_T VAL, UINT8_T NRDIG, UINT8_T
 FLAGS)
 VOID UART1_WRITEHEX32(UINT32_T VAL, UINT8_T NRDIG, UINT8_T
 FLAGS)

uart0_writehex16 / uart1_writehex16 and uart0_writehex32 / uart1_writehex32 write a hexadecimal number to the UART. The number of desired digits must be given in the nrdig parameter. It is blocking if the number of digits exceed the FIFO free space. flags is a bitwise or combination of the WRNUM constants documented in section 3.3.19. **Error! Bookmark not defined.**

3.6.7. VOID UART0_WRITENUM16(UINT16_T VAL, UINT8_T NRDIG,
 UINT8_T FLAGS)
 VOID UART0_WRITENUM32(UINT32_T VAL, UINT8_T NRDIG, UINT8_T
 FLAGS)

```

VOID UART1_WRITENUM16(UINT16_T VAL, UINT8_T NRDIG, UINT8_T
  FLAGS)
VOID UART1_WRITENUM32(UINT32_T VAL, UINT8_T NRDIG, UINT8_T
  FLAGS)

```

uart0_writenum16 / uart1_writenum16 and uart0_writenum32 / uart1_writenum32 write a hexadecimal number to the UART. The number of desired digits must be given in the nrdig parameter. It is blocking if the number of digits exceed the FIFO free space. *flags* is a bitwise or combination of the WRNUM constants documented in section **Error! Bookmark not defined.**

```

3.6.8.    UINT8_T UART0_TXBUFFERSIZE(VOID)
          UINT8_T UART1_TXBUFFERSIZE(VOID)
          UINT8_T UART0_RXBUFFERSIZE(VOID)
          UINT8_T UART1_RXBUFFERSIZE(VOID)

```

These functions returns transmit and receive buffer sizes. Buffer sizes are configurable.

```

3.6.9.    UINT8_T UART0_RXCOUNT(VOID)
          UINT8_T UART1_RXCOUNT(VOID)

```

uart0_rxcount / uart1_rxcount returns the number of characters in the receive FIFO.

```

3.6.10.   UINT8_T UART0_TXFREE(VOID)
          UINT8_T UART1_TXFREE(VOID)

```

uart0_txfree / uart1_txfree returns the number of free space for characters in the transmit FIFO.

```

3.6.11.   UINT8_T UART0_TXIDLE(VOID)
          UINT8_T UART1_TXIDLE(VOID)

```

uart0_txidle / uart1_txidle returns one if the transmitter is idle, i.e. all transmit characters have left the wire. This routine can be used to determine whether the microprocessor can enter sleep.

```

3.6.12.   VOID UART0_WAIT_RXCOUNT(UINT8_T V)
          VOID UART1_WAIT_RXCOUNT(UINT8_T V)

```

uart0_wait_rxcount / uart1_wait_rxcount blocks until the number of characters in the receive FIFO reach or exceed v.

```

3.6.13.   VOID UART0_WAIT_TXDONE(VOID)
          VOID UART1_WAIT_TXDONE(VOID)

```

uart0_wait_txdone / uart1_wait_txdone blocks until the last character in the transmit FIFO has been transmitted.

3.6.14. VOID UART0_WAIT_TXFREE(UINT8_T V)
 VOID UART1_WAIT_TXFREE(UINT8_T V)

uart0_wait_txfree / uart1_wait_txfree blocks until the number of free space for characters in the transmit FIFO reaches or exceeds v.

3.6.15. VOID UART0_RXADVANCE(UINT8_T IDX)
 VOID UART1_RXADVANCE(UINT8_T IDX)

uart0_rxadvance / uart1_rxadvance drops the given number of characters from the front of the FIFO.

3.6.16. VOID UART0_TXADVANCE(UINT8_T IDX)
 VOID UART1_TXADVANCE(UINT8_T IDX)

uart0_txadvance / uart1_txadvance adds the given number of characters at the end of the transmit FIFO. They must have been defined by uart0_txpoke / uart1_txpoke or uart0_txpokehex / uart1_txpokehex before, otherwise the transmitted characters are undefined.

3.6.17. UINT8_T UART0_RXPEEK(UINT8_T IDX)
 UINT8_T UART1_RXPEEK(UINT8_T IDX)

uart0_rxpeek / uart1_rxpeek returns the idx'th character in the receive FIFO without modifying the FIFO.

3.6.18. VOID UART0_TXPOKE(UINT8_T IDX, UINT8_T CH)
 VOID UART1_TXPOKE(UINT8_T IDX, UINT8_T CH)

uart0_txpoke / uart1_txpoke puts a character at the idx'th position after the end of the transmit FIFO. It does not become part of the FIFO. In order to actually transmit the character, uart0_txadvance / uart1_txadvance must be called.

3.6.19. VOID UART0_TXPOKEHEX(UINT8_T IDX, UINT8_T CH)
 VOID UART1_TXPOKEHEX(UINT8_T IDX, UINT8_T CH)

uart0_txpoke / uart1_txpoke puts a hexadecimal character at the idx'th position after the end of the transmit FIFO. It does not become part of the FIFO. In order to actually transmit the character, uart0_txadvance / uart1_txadvance must be called.

3.6.20. UINT8_T UART0_POLL(VOID)
 UINT8_T UART1_POLL(VOID)

Normally, data is transferred between the FIFO and the hardware using an interrupt handler. Sometimes though, an interrupt handler is undesirable; in this case, uart0_poll / uart1_poll can be called periodically to transfer data between the FIFO and the hardware if available. It returns a bit mask with bit 0 set if a byte was transferred from the hardware to the receive FIFO, and bit 1 set if a byte was transferred from the transmit FIFO to the hardware.

3.6.21. UART0_DEFINE_TXBUFFER(sz)
 UART1_DEFINE_TXBUFFER(sz)
 UART0_DEFINE_RXBUFFER(sz)
 UART1_DEFINE_RXBUFFER(sz)

These macros defines transmit and receive buffer sizes, respectively. The argument needs to lie between 2 and 256 (inclusive). Note that the argument specifies the total buffer size; one character is unusable due to the design of the buffer pointers, therefore `uart0_txbuffer_size / uart1_txbuffer_size` and `uart0_rxbuffer_size / uart1_rxbuffer_size` will return one byte less than the argument. These Macros should be defined outside of all the functions.

3.7. LIBMFFLASH.H

3.7.1. VOID FLASH_UNLOCK(VOID)

`flash_unlock` prepares the FLASH controller for subsequent erase or write operations.

3.7.2. VOID FLASH_LOCK(VOID)

`flash_lock` disables further erase and write operations until `flash_unlock` is called again. This provides some protection against accidental FLASH modifications.

3.7.3. INT8_T FLASH_PAGEERASE(UINT16_T PGADDR)

AX8052:

FLASH memory is organized as 64 sectors of 1kByte. Each sector may be erased individually. Erasing a sector sets it to all ones. An arbitrary address located inside the desired sector must be given as `pgaddr`. `flash_pageerase` only succeeds if the flash controller has been enabled previously, using `flash_unlock`. It returns zero on success and nonzero on error.

AXM0F243:

FLASH memory 64KB is organized as 512 pages of 128 bytes each. Each page may be erased individually. Erasing a page sets it to all zeros. An arbitrary address located inside the desired page must be given as `pgaddr`. `flash_pageerase` only succeeds if the flash controller has been enabled previously, using `flash_unlock`. It returns zero on success and nonzero on error.

3.7.4. INT8_T FLASH_WRITE(UINT16_T WADDR, UINT16_T WDATA)

AX8052:

`flash_write` writes the 16 bit word `wdata` to the given address `waddr`. The AX8052 FLASH is organized as 16 bit words. `waddr` must be an even address. `flash_write` can only program FLASH bits from 1 to 0, therefore the respective FLASH sector should be erased before calling write. `flash_write` only succeeds if the flash controller has been enabled previously, using `flash_unlock`. It returns zero on success and nonzero on error.

AXM0F243:

flash_write writes the 16 bit word wdata to the given address waddr. waddr must be an even address. flash_write can only program FLASH bits from 0 to 1, therefore the respective FLASH page should be erased before calling write. flash_write only succeeds if the flash controller has been enabled previously, using flash_unlock. It returns zero on success and nonzero on error.

3.7.5. `UINT16_T FLASH_READ(UINT16_T RADDR)`

flash_read is a convenience function to read from FLASH. Reading from FLASH can also be accomplished by dereferencing a `__code *`.

3.7.6. `UINT8_T FLASH_APPLY_CALIBRATION(VOID)`

AX8052:

The last FLASH sector from address 0xFC00 to 0xFFFF should not be used by the user. It may store factory calibration data. flash_apply_calibration writes the calibration data to the respective chip registers if found. It should be called very early during program initialization.

Before calling flash_apply_calibration, the correct LPOSC operating mode should be set (Bit LPOSCFAST of Register LPOSCCONFIG); different calibration values are used for both modes.

flash_apply_calibration returns a nonzero value if valid calibration data is found.

AXM0F243:

This API is not available.

3.8. LIBMFRADIO.H

libmfradio.h contains access, initialization and configuration routines for the radio peripheral.

3.8.1. `UINT8_T RADIO_READ8(UINT16_T ADDR)`
 `UINT16_T RADIO_READ16(UINT16_T ADDR)`
 `UINT32_T RADIO_READ24(UINT16_T ADDR)`
 `UINT32_T RADIO_READ32(UINT16_T ADDR)`

These routines read radio registers using 8-bit, 16-bit, 24-bit or 32-bit (plus address) SPI transactions.

3.8.2. `VOID RADIO_WRITE8(UINT16_T ADDR, UINT8_T D)`
 `VOID RADIO_WRITE16(UINT16_T ADDR, UINT16_T D)`
 `VOID RADIO_WRITE24(UINT16_T ADDR, UINT32_T D)`
 `VOID RADIO_WRITE32(UINT16_T ADDR, UINT32_T D)`

These routines write radio registers using 8-bit, 16-bit, 24-bit or 32-bit (plus address) SPI transactions.


```
3.8.3.  UINT8_T AX5031_RESET(VOID)
        UINT8_T AX5042_RESET(VOID)
        UINT8_T AX5043_RESET(VOID)
        UINT8_T AX5044_45_RESET(VOID)
        UINT8_T AX5051_RESET(VOID)
```

These functions probe for the respective radio peripheral and reset it if found. They return zero if the respective radio peripheral is found, or a nonzero error code. These routines work both for the SoC devices, as well as the two-chip combination of AX8052F100/AXM0F243 plus the respective ON Semiconductor Radio Chip. There are small differences between the SoC and the two chip solution; these routines automatically detect and handle those differences.

```
3.8.4.  VOID AX5031_COMMSLEEPEXIT(VOID)
        VOID AX5042_COMMSLEEPEXIT(VOID)
        VOID AX5043_COMMSLEEPEXIT(VOID)
        VOID AX5044_45_COMMSLEEPEXIT(VOID)
        VOID AX5051_COMMSLEEPEXIT(VOID)
```

These functions (re)initialize the communication with the radio peripheral. They should be called after waking up the microcontroller from sleep or deep sleep, unless the radio was also in deep sleep, in which case `axxxx_wakeup_deepsleep()` should be called. During cold boot, `axxxxx_reset()` should be called instead.

```
3.8.5.  VOID AX5031_COMMINIT(VOID)
        VOID AX5042_COMMINIT(VOID)
        VOID AX5043_COMMINIT(VOID)
        VOID AX5044_45_COMMINIT(VOID)
        VOID AX5051_COMMINIT(VOID)
```

These functions perform basic (re)initialization of the radio communication interface. They do not set up interrupt routing. These routines are infrequently used.

```
3.8.6.  VOID AX5031_RCLK_ENABLE(UINT8_T DIV)
        VOID AX5042_RCLK_ENABLE(UINT8_T DIV)
        VOID AX5043_RCLK_ENABLE(UINT8_T DIV)
        VOID AX5044_45_RCLK_ENABLE(UINT8_T DIV)
        VOID AX5051_RCLK_ENABLE(UINT8_T DIV)
```

These functions enable the radio system clock RSYCLK. The `div` argument can be within the range 0...11. It causes the radio system clock to be divided by 2^{div} . After calling this function, the radio system clock may be used by any of the AX8052/AXM0F243 core peripherals, such as the general purpose timers, the wakeup timers, or even as the system clock.

Often, the radio peripheral has the most accurate clock (crystal or even TCXO derived). These functions allow the precise clock to be used for other purposes within the microcontroller. On the other hand, RSYCLK should not be enabled needlessly to save energy.

```
3.8.7.    VOID AX5042_RCLK_DISABLE(VOID)
          VOID AX5031_RCLK_DISABLE(VOID)
          VOID AX5043_RCLK_DISABLE(VOID)
          VOID AX5044_45_RCLK_DISABLE(VOID)
          VOID AX5051_RCLK_DISABLE(VOID)
```

These functions disable the radio system clock RSYCLK forwarding to the microcontroller. It may still be used within the radio peripheral.

```
3.8.8.    VOID AX5043_RCLK_WAIT_STABLE(UINT8_T WTFLAG)
          VOID AX5044_45_RCLK_WAIT_STABLE(UINT8_T WTFLAG)
```

This function can be invoked after enabling the radio clock (RSYCLK) to wait for RSYCLK to become stable before using it. *wtflag will dictate chip to stay in which low power mode while waiting for RSYCLK to stabilize. wtflag can be any of the low power mode macro, WTFLAG_CANSLEEP, WTFLAG_CANSTANDBY or WTFLAG_CANSLEEPCONT.*

```
3.8.9.    VOID AX5043_ENTER_DEEPSLEEP(VOID)
          VOID AX5044_45_ENTER_DEEPSLEEP(VOID)
```

This function puts the radio into deep sleep mode. Note that the radio power states are distinct from the microcontroller core power states.

```
3.8.10.   UINT8_T AX5043_WAKEUP_DEEPSLEEP(VOID)
          UINT8_T AX5044_45_WAKEUP_DEEPSLEEP(VOID)
```

This function wakes the radio up from deep sleep. Note that the radio power states are distinct from the microcontroller core power states.

```
3.8.11.   VOID AX5043_READFIFO(UINT8_T *PTR, UINT8_T LEN)
          VOID AX5044_45_READFIFO(UINT8_T *PTR, UINT8_T LEN)
```

This functions reads a given number of bytes from the FIFO and stores it at the given location.

```
3.8.12.   VOID AX5043_WRITEFIFO(CONST UINT8_T *PTR, UINT8_T LEN)
          VOID AX5044_45_WRITEFIFO(CONST UINT8_T *PTR, UINT8_T LEN)
```

This functions writes a given number of bytes at the pointer to the FIFO.

```
3.8.13.   VOID AX5043_SET_PWRAMP_PIN(UINT8_T CONFIG_PIN)
          VOID AX5044_45_SET_PWRAMP_PIN(UINT8_T CONFIG_PIN)
```

This API configures the edge trigger interrupt on PWRAMP pin (P2.3) and configures strong drive mode for the *config_pin* to which the PWRAMP pin (P2.3) functionality will be forwarded if the input parameter is any of the GPIO port pin value from the GPIO_PIN_Enum defined in the axm0f2_pin.h header file.

If *config_pin* is GPIO_PIN_DISABLE, the interrupt on PWRAMP pin (P2.3) will be disabled.

If *config_pin* is GPIO_PIN_INVERT OR'ed with any of the GPIO Pin enum value, then the functionality on the *config_pin* will be inverted.

```
3.8.14.    UINT8_T AX5043_GET_PWRAMP_PIN()
           UINT8_T AX5044_45_GET_PWRAMP_PIN()
```

This API will return the GPIO port pin *config_pin*, enum value on which the PWRAMP pin (P2.3) functionality is available. The *ax5043_set_pwramp_pin()* function should be called before calling this function.

```
3.8.15.    VOID AX5043_SET_ANTSEL_PIN(UINT8_T CONFIG_PIN)
           VOID AX5044_45_SET_ANTSEL_PIN(UINT8_T CONFIG_PIN)
```

This API configures the edge trigger interrupt on ANTSEL pin (P2.6) and configures strong drive mode for the *config_pin* to which the ANTSEL pin (P2.6) functionality will be forwarded if the input parameter is any of the GPIO port pin value from the GPIO_PIN_Enum defined in the *axm0f2_pin.h* header file.

If *config_pin* is GPIO_PIN_DISABLE, the interrupt on ANTSEL pin (P2.6) will be disabled.

If *config_pin* is GPIO_PIN_INVERT OR'ed with any of the GPIO Pin enum value, then the functionality on the *config_pin* will be inverted.

```
3.8.16.    UINT8_T AX5043_GET_ANTSEL_PIN()
           UINT8_T AX5044_45_GET_ANTSEL_PIN()
```

This API will return the GPIO port pin *config_pin*, enum value on which the ANTSEL pin (P2.6) functionality is available. The *ax504xxx_set_antsel_pin()* function should be called before calling this function.

3.9. LIBMFADC.H

libmfadc.h contains utility functions for use with the on-chip Analog to Digital Converter (ADC).

```
3.9.1.    INT16_T ADC_MEASURE_TEMPERATURE(VOID)
```

This function measures the temperature of the microcontroller die. With the default calibration (after calling *flash_apply_calibration()*), the function returns whole degrees celsius in the upper byte, and fractional degrees in the lower byte. The temperature is therefore return value / 256 °C.

This function sets up the A/D controller for temperature conversion, and restores the old settings afterwards. It disables all interrupts during conversion to reduce noise on the conversion result. One conversion takes approximately 500µs for AX8052 and 25µs for AXM0F243.

3.9.2. UINT16_T ADC_SINGLEENDED_OFFSET_X1(VOID)

This API is applicable only for AX8052, in single ended $\times 1$ mode, the ADC exhibits a small offset. It is recommended to cancel the offset by adding the value returned by this function to all ADC single ended $\times 1$ mode results. The return value of this function is only valid if called after `flash_apply_calibration()` has been called.

3.10. LIBMFWTIMER.H

`libmfwtimer.h` provides sophisticated timer event handling. It manages both timers. Timer 0 is intended to be clocked with one of the low power clocks, i.e. LPOSC or, if an external tuning fork crystal is connected, with LPXOSC. Its time is kept even during sleep periods. In contrast, Timer 1 is intended to time shorter intervals. It is clocked by one of the faster clocks (such as FRCOSC, XOSC or the radio clock), but its time is not kept during sleep.

The wakeup timer module keeps times as 32 bit integers with wrap-around. This results in events that can be scheduled approximately 2^{30} in the future. The absolute time depends on the clock frequency and the prescaler setting of the timer.

In order to schedule a timer event, the user has to allocate a `wtimer_desc` structure in xdata memory, write the address of a handler function into the `wtimer_desc` structure, write the expiry time, and call `wtimer{0,1}_add{absolute,relative}` function, supplying the address of the descriptor. The wakeup timer module then keeps a linked list of descriptors, sorted by expiry time.

When the time comes, the handler is not called directly from the interrupt handler; doing so would have the following disadvantages:

- Interrupt handlers should be short running, since other interrupts of the same priority level are blocked while an interrupt handler runs
- Handler functions would need to be declared reentrant, and also all the routines it calls. Failure to do so would result in hard to find bugs.

Instead, the descriptor is moved to the end of a third queue, the pending queue. Handlers in the pending queue are called as soon as `wtimer_runcallbacks` is called. This mechanism is also available for other interrupt handlers by using `wtimer_add_callback`.

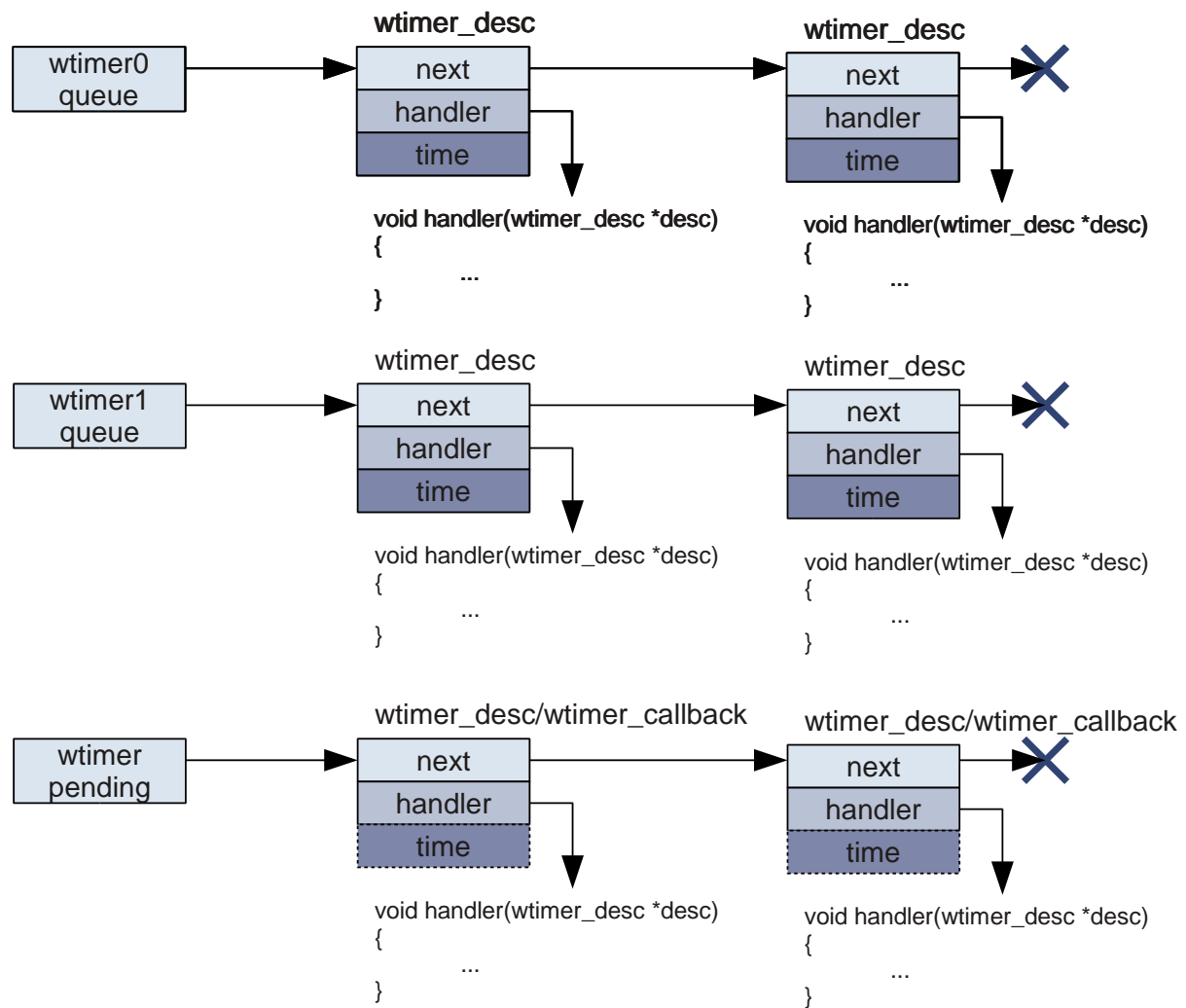


Figure 1: Wakeup Timer Queues

WCOWDT is used as source for WTIME0 and TCPWM4 is used as source for WTIMER1.

AXM0F243: Total five, TCPWM 0 to 4 timer/counter/pwm 's are available in AXM0F243. TCPWM0 and TCPWM4 are used for libmf system features. TCPWM0, TCPWM1 and TCPWM2 are available for application use.

The main loop should repeatedly call wtimer_idle. The program structure should look like this:

```
void main(void)
{
    wtimer0_setclksrc(...);
    wtimer1_setclksrc(...);
    flash_apply_calibration();
    wtimer_init(...);
    ...
    for (;;) {
        uint8_t flg;
```

```

        wtimer_runcallbacks();
        flg = WTFLAG_CANSTANDBY;
        if (dbglink_txidle() && ...)
            flg |= WTFLAG_CANSLEEP;
        wtimer_idle(flg);
    }
}

```

When using SDCC/ARM-GCC, libmfwtimer.h must be included in the file containing the definition for main.

```

3.10.1.    UINT32_T WTIMER0_CURTIME(VOID)
           UINT32_T WTIMER1_CURTIME(VOID)

```

These functions retrieve the current time of the respective timer.

```

3.10.2.    VOID WTIMER0_ADDABSOLUTE(__XDATA STRUCT WTIMER_DESC
           *DESC)
           VOID WTIMER1_ADDABSOLUTE(__XDATA STRUCT WTIMER_DESC *DESC)
           VOID WTIMER0_ADDRELATIVE(__XDATA STRUCT WTIMER_DESC *DESC)
           VOID WTIMER1_ADDRELATIVE(__XDATA STRUCT WTIMER_DESC *DESC)

```

These functions add a descriptor to the queue of the respective timer (maintaining the ascending time property of the queue). The relative versions first add the current time to the time field of the descriptor. Care must be taken not to re-add a descriptor already in a queue (call wtimer{0,1}_remove first if in doubt).

```

3.10.3.    UINT8_T WTIMER0_REMOVE(__XDATA STRUCT WTIMER_DESC *DESC)
           UINT8_T WTIMER1_REMOVE(__XDATA STRUCT WTIMER_DESC *DESC)
           UINT8_T WTIMER_REMOVE(__XDATA STRUCT WTIMER_DESC *DESC)

```

Remove a descriptor from the respective timer's queue or the pending queue. The handler is not called. These functions return whether the descriptor was found in any of the queues. wtimer_remove examines both timer queues.

```

3.10.4.    VOID WTIMER_ADD_CALLBACK(__XDATA STRUCT WTIMER_CALLBACK
           *DESC)

```

Add a callback descriptor to the pending queue. Care must be taken not to re-add a descriptor already in the pending queue. If in doubt, call wtimer_remove_callback first.

```

3.10.5.    UINT8_T WTIMER_REMOVE_CALLBACK(__XDATA STRUCT
           WTIMER_CALLBACK *DESC)

```

Remove a callback descriptor from the pending queue. Returns whether the descriptor was found in the pending queue. The handler is not called.

3.10.6. `UINT8_T WTIMER_IDLE(UINT8_T FLAGS)`

Send the microprocessor to standby or sleep, unless there are pending timer or callback events. The processor will only go to sleep if `WTFLAG_CANSLEEP` or `WTFLAG_CANSLEEPCONT` is set in the flags argument, and if there are no timer events set on wakeup timer 1. Otherwise it will go into standby mode if `WTFLAG_CANSTANDBY` is set in the flags argument. The function returns `WTIDLE_WORK` if there are pending timer events or callbacks, `WTIDLE_SLEEP` if `enter_sleep_cont()` was called and peripherals should now be reinitialized, and zero otherwise. Before calling `wtimer_idle`, `wtimer_runcallbacks` should be called.

3.10.7. `UINT8_T WTIMER_RUNCALLBACKS(VOID)`

This function runs all pending timer and callback events. It returns the number of callbacks called.

3.10.8. `UINT8_T WTIMER_CANSLEEP(VOID)`

This function returns one if the microprocessor can safely go to sleep mode. It returns zero if the timer 1 queue is not empty. Furthermore, the SDCC/ARM-GCC version also returns zero if any of the used UARTs or DebugLink (if used) still has transmit data queued.

3.10.9. `VOID WTIMER0_SETCLKSRC(UINT8_T CLKSRC, UNIT8_T PRESCALER)` `VOID WTIMER1_SETCLKSRC(UINT8_T CLKSRC, UNIT8_T PRESCALER)`

AX8052:

These functions set the clock sources for both wakeup timers. These functions should be called as early as possible; when using SDCC, call them from `_sdcc_external_startup`. These functions take care not to accidentally enable a crystal oscillator when switching sources.

For the first argument, use one of the constants `CLKSRC_FRCOSC`, `CLKSRC_LPOSC`, `CLKSRC_XOSC`, `CLKSRC_LPXOSC`, `CLKSRC_RSYSCLK`, `CLKSRC_TCLK`, `CLKSRC_SYSCLK`, `CLKSRC_OFF`.

For the second argument, 0 means $\times 2$, 1 means $\times 1$, 2 means $\div 2$, 3 means $\div 4$, ... and 7 means $\div 64$.

AXM0F243:

These functions set the clock sources for both wakeup timers. These functions should be called as early as possible; Call them from `_axm0f2_external_startup`. These functions take care not to accidentally enable a crystal oscillator when switching sources.

`void wtimer0_setclksrc(uint8_t clksrc, unit8_t prescaler)`

clksrc can be one of the constant `CLKSRC_LPOSC` or `CLKSRC_LPXOSC`.

Prescaler will be ignored and always treated as one.

`void wtimer1_setclksrc(uint8_t clksrc, unit8_t prescaler)`

clksrc will be ignored and always treated as `CLKSRC_FRCOSC` (IMO).

Prescaler can be 0 to 7. Clock source will be divided by $2^{\text{Prescaler}}$.

3.10.10. VOID WTIMER_INIT(VOID)

This function initializes the wakeup timer core. It must be called before any other wakeup timer function, except `wtimer0/1_setclsrc`, can be called. When waking up from deep sleep, `wtimer_init_deepsleep` must be called instead.

3.10.11. VOID WTIMER_INIT_DEEPSLEEP(VOID)

This function performs initialization of the wakeup timer core, like `wtimer_init`, but for the case of waking up from deep sleep.

3.11. LIBMFOSC.H

This section is applicable for AX8052 only.

`libmfosc.h` contains oscillator related utility functions.

3.11.1. VOID TURN_OFF_XOSC(VOID)

This function turns the crystal oscillator off if it has been accidentally enabled, but no crystal is connected to its pins. This function needs to drive and toggle pins PA0 and PA1, so its usability depends on the target board design.

3.11.2. VOID TURN_OFF_LPXOSC(VOID)

This function turns the low power crystal oscillator off if it has been accidentally enabled, but no crystal is connected to its pins. This function needs to drive and toggle pins PA4 and PA5, so its usability depends on the target board design.

3.11.3. VOID SETUP_XOSC(VOID)

This function sets up the I/O pins and other configuration suitably so that the crystal oscillator may run. Call this function if a crystal is connected to the pins PA0/PA1.

3.11.4. VOID SETUP_LPXOSC(VOID)

This function sets up the I/O pins and other configuration suitably so that the low power crystal oscillator may run. Call this function if a crystal is connected to the pins PA3/PA4.

3.11.5. UINT8_T SETUP_OSC_CALIBRATION(UINT32_T REFFREQ, UINT8_T REFOSC)

AX8052:

This function sets up the low power and the fast RC oscillator configuration circuit for calibration from a given reference frequency and source. The parameter `refosc` specifies the oscillator to use as a reference. Valid values are `CLKSRC_XOSC`, `CLKSRC_LPXOSC` and `CLKSRC_RSYSCLK`. `reffreq` specifies the frequency of the reference source in Hz. The reference frequency must lie between 500Hz and 32.768MHz. For ON Semiconductor

Transceivers with a faster crystal or TCXO (such as the AX8052F143 with a 48MHz TCXO), it is recommended to divide SYSCCLK in the transceiver block to obtain a reference clock within range (for example with the AX5043_PINFUNCSYSCCLK register). This function returns zero on success, or non-zero if the input parameters are out of range.

AXM0F243:

This function sets up the low power clock ILO (internal low-speed oscillator) and IMO (internal main oscillator) configuration circuit for calibration from a given reference frequency and source. The parameter *refosc* specifies the oscillator to use as a reference. Current implementation supports CLKSRC_RSYSCCLK. *reffreq* specifies the frequency of the reference source in Hz. The reference frequency must lie between 4MHz and 48MHz. This function returns zero on success, or non-zero if the input parameters are out of range.

3.11.6. SETUP_OSC_CALIBRATION_CONST(REFFREQ,REFOSC)

Available only for AX8052 and This is a macro version of the setup_osc_calibration function. It may be used if both reffreq and refosc are compile time constants (which they often are). It can be simplified by the compiler to register writes with constants, therefore producing very little code. If one or more of the parameters is outside the valid range, calibration is not set up and no error is returned. For the parameter valid range, see setup_osc_calibration.

3.11.7. UINT32_T WTIMER0_CORRECTINTERVAL(UINT32_T INTVL) UINT32_T WTIMER1_CORRECTINTERVAL(UINT32_T INTVL)

AX8052:

This function is a dummy implementation for the AX8052 to support backward compatibility that returns the same value as passed, *intvl*.

AXM0F243:

This function returns corrected/calibrated interval for the timer, for given *intvl*, if the clock source for the timer is internal low power clock (CLKSRC_LPOSC) or returns the same value *intvl* if the clock source for the timer is other than CLKSRC_LPOSC. setup_osc_calibration function should be called before calling these functions.

3.12. CRITICAL SECTION

These API are implemented to manage critical sections.

Data type "typedef uint32_t criticalsection_t" is declared to use with critical section APIs.

3.12.1. CRITICALSECTION_T ENTER_CRITICAL(VOID)

This API stores interrupt register content and disables all user interrupts.

3.12.2. VOID EXIT_CRITICAL(CRITICALSECTION_T CRIT)

This API restores interrupt register content that is stored in enter_critical() function.

3.12.3. VOID REENTER_CRITICAL(VOID)

This API disables all user interrupts.

3.13. SPI DRIVER

AXM0F243:

These driver APIs are implemented to support SPI transfer of data to and from the SPI bus. These APIs are exposed through 'libmfspi.h'. This driver supports only master mode.

3.13.1. UINT8_T SPI0_INIT(UINT8_T SPEED, UINT8_T SEL_DIV, UINT8_T MODE, UINT8_T BITORDER)

Description: This API initializes SPI on SCB0 for the below user specified configurations.

Speed: SPI clock speed can be any of below

LIBMF_SPI_SPEED_1MHZ
LIBMF_SPI_SPEED_2MHZ
LIBMF_SPI_SPEED_3MHZ
LIBMF_SPI_SPEED_4MHZ
LIBMF_SPI_SPEED_5MHZ
LIBMF_SPI_SPEED_6MHZ
LIBMF_SPI_SPEED_7MHZ
LIBMF_SPI_SPEED_8MHZ

sel_div: Integer divider can be any of the below

LIBMF_SPI_INT_DIV_0
LIBMF_SPI_INT_DIV_1
LIBMF_SPI_INT_DIV_2
LIBMF_SPI_INT_DIV_3
LIBMF_SPI_INT_DIV_4
LIBMF_SPI_INT_DIV_5

mode: Clock polarity and clock phase can be any of the below

LIBMF_SPI_CPOLO_CPHA0
LIBMF_SPI_CPOLO_CPHA1
LIBMF_SPI_CPOL1_CPHA0
LIBMF_SPI_CPOL1_CPHA1

Bitorder: Bit order can be any of the below

LIBMF_SPI_LSB_FIRST
LIBMF_SPI_MSB_FIRST

3.13.2. VOID SPI0_READ(UINT8_T *RDBUF, UINT8_T RDLLENGTH)

Enables read related interrupts and updates spi_status to AXM0_STATUS_IN_PROGRESS. Reads from spi bus (on SCB0) to rdbuf for specified rdlength in interrupt context. Function returns immediately but rdbuf will be used in interrupt context to read data. Data will be available in rdbuf only when spi status becomes AXM0_STATUS_SUCCESS. Refer usage examples below.

rdbuf: User provided buffer pointer to store the data read
rdlength: Number of bytes to be read

3.13.3. VOID SPI0_WRITE(UINT8_T *WRBUF, UINT8_T WRLENGTH)

Enables write related interrupts and updates spi_status to AXM0_STATUS_IN_PROGRESS. Writes data from wrbuf to spi bus (on SCB0) for specified wrlength in interrupt context. Function returns immediately but wrbuf will be used in interrupt context to write data. Data has been completely sent out from the wrbuf only when spi status becomes AXM0_STATUS_SUCCESS.

wrbuf: User provided buffer pointer containing data to be written
wrlength: Number of bytes to be written

3.13.4. UINT8_T SPI0_WAIT_UNTIL_DONE(VOID)

Returns spi status if SPI is done with the transfer else put chip into standby mode and wait for the interrupt to wake-up. This is a blocking function.

3.13.5. UINT8_T SPI0_GET_STATUS(VOID)

Returns spi status.

spi_status can be any of the following.

AXM0_STATUS_SUCCESS
AXM0_STATUS_FAIL
AXM0_STATUS_INVALID_PARAMETER
AXM0_STATUS_IN_PROGRESS
AXM0_STATUS_NO_INIT

3.13.6. VOID SPI0_CLOSE(VOID)

Disables SPI and returns status.

Usage example 1: SPI write - blocking for the status

```
uint8_t spi_status = spi0_init(LIBMF_SPI_SPEED_1MHZ, LIBMF_SPI_INT_DIV_0,
LIBMF_SPI_CPOLO_CPHA0, LIBMF_SPI_MSB_FIRST); /* Speed 1 MHz, Integer divider number,
Mode 0, MSB first */

If(spi_status == AXM0_STATUS_SUCCESS)
{
    uint8_t wtbuf[] = {user data}; /* This buffer will be used for spi data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    spi0_write(wtbuf, data_len);

    spi_status = spi0_wait_until_done();

    If(spi_status == AXM0_STATUS_SUCCESS)
    {
        /* spi finished with the transfer */
    }
}
```

Usage example 2: SPI read - blocking for the status

```
uint8_t spi_status = spi0_init(LIBMF_SPI_SPEED_1MHZ, LIBMF_SPI_INT_DIV_0,
LIBMF_SPI_CPOLO_CPHA0, LIBMF_SPI_MSB_FIRST); /* Speed 1 MHz, Integer divider number,
Mode 0, MSB first */

If(spi_status == AXM0_STATUS_SUCCESS)
{
    uint8_t rdbuf[n]; /* This buffer will be used for spi data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    spi0_read(rdbuf, data_len);

    spi_status = spi0_wait_until_done();
    If(spi_status == AXM0_STATUS_SUCCESS)
    {
        /* rdbuf contains SPI read data */
        /* spi finished with the transfer */
    }
}
```

Usage example 3: SPI write - non-blocking

```

uint8_t spi_status = spi0_init(LIBMF_SPI_SPEED_1MHZ, LIBMF_SPI_INT_DIV_0,
LIBMF_SPI_CPOLO_CPHA0, LIBMF_SPI_MSB_FIRST); /* Speed 1 MHz, Integer divider number,
Mode 0, MSB first */

If(spi_status == AXM0_STATUS_SUCCESS)
{
    uint8_t wrbuf[] = {user data}; /* This buffer will be used for spi data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    spi0_write(wrbuf, data_len);

    while(spi0_get_status() == AXM0_STATUS_IN_PROGRESS)
    {
        /* Do user tasks if any */
    }
    /* spi finished with the transfer */
}

```

Usage example 4: SPI read - non-blocking

```

uint8_t spi_status = spi0_init(LIBMF_SPI_SPEED_1MHZ, LIBMF_SPI_INT_DIV_0,
LIBMF_SPI_CPOLO_CPHA0, LIBMF_SPI_MSB_FIRST); /* Speed 1 MHz, Integer divider number,
Mode 0, MSB first */

If(spi_status == AXM0_STATUS_SUCCESS)
{
    uint8_t rdbuf[n]; /* This buffer will be used for spi data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    spi0_read(rdbuf, data_len);

    while(spi0_get_status() == AXM0_STATUS_IN_PROGRESS)
    {
        /* Do user tasks if any */
    }
    /* rdbuf contains SPI read data */
    /* spi finished with the transfer */
}

```

3.14. I2C DRIVER

AXM0F243:

These driver APIs are implemented to support I2C transfer of data to and from the I2C bus. These APIs are exposed through 'libmfi2cm.h'. This driver supports only master mode.

```
3.14.1.    UINT8_T I2C0_INIT(UINT8_T MODE, UINT8_T SPEED, UINT8_T
            SEL_DIV, UINT8_T CLK_SRC)
            UINT8_T I2C1_INIT(UINT8_T MODE, UINT8_T SPEED, UINT8_T
            SEL_DIV, UINT8_T CLK_SRC)
```

This API initializes I2C0 on SCB0 and I2C1 on SCB2 for the below user specified configurations.

Mode: I2C mode

LIBMF_I2C_MASTER (Supports only master mode)

Speed: I2C clock speed can be any of below

LIBMF_I2C_SLOW_MODE : 50kbps
 LIBMF_I2C_STANDARD_MODE : 100kbps
 LIBMF_I2C_FAST_MODE : 400kbps
 LIBMF_I2C_FAST_PLUS_MODE : 1mbps

sel_div: Integer divider can be any of the below

LIBMF_I2C_INT_DIV_0
 LIBMF_I2C_INT_DIV_1
 LIBMF_I2C_INT_DIV_2
 LIBMF_I2C_INT_DIV_3
 LIBMF_I2C_INT_DIV_4
 LIBMF_I2C_INT_DIV_5

clk_src:

LIBMF_I2C_INTERNAL_CLK (Supports only internal clock)

```
3.14.2.    VOID I2C0_READ(UINT16_T SLAVEADDRESS, UINT8_T *RDATA,
            UINT8_T LENGTH, UINT8 REPEATED)
            VOID I2C1_READ(UINT16_T SLAVEADDRESS, UINT8_T *RDATA,
            UINT8_T LENGTH, UINT8 REPEATED)
```

Enables read related interrupts and updates i2c_status to AXM0_STATUS_IN_PROGRESS. Reads from i2c0 bus (on SCB0) or i2c1 bus (on SCB2) to rddata for specified length in the interrupt context with user specified repeated mode. Function returns immediately but rddata will be used in interrupt context to read the data. Data will be available in rddata only when i2c status becomes AXM0_STATUS_SUCCESS

slaveAddress: Address of slave to communicate
 (Example: If Slave address is 0x7E,
 7 bit addressing: 0x7E
 10 bit addressing: 0xF07E (User should append 0xF in MSB for 10 bit mode))

rdData: Read data buffer

length: Number of bytes to be read from the slave

repeated: Repeated start

0: Stop bit is generated at the end

1: Stop bit is not generated at the end

```
3.14.3.    VOID I2C0_WRITE(UINT16_T SLAVEADDRESS, UINT8_T *WRDATA,
            UINT8_T LENGTH, UINT8 REPEATED)
            VOID I2C1_WRITE(UINT16_T SLAVEADDRESS, UINT8_T *WRDATA,
            UINT8_T LENGTH, UINT8 REPEATED)
```

Enables write related interrupts and updates i2c_status to AXM0_STATUS_IN_PROGRESS. Writes data from wrdata to i2c0 bus (on SCB0) or i2c1 bus (on SCB2) for specified length in the interrupt context with specified repeated mode. Function returns immediately but wrdata will be used in interrupt context to write data. Data has been sent out from the wrdata only when i2c status becomes AXM0_STATUS_SUCCESS.

slaveAddress: Address of slave to communicate

(Example: If Slave address is 0x7E,

7 bit addressing: 0x7E

10 bit addressing: 0xF07E (User should append 0xF in MSB for 10 bit mode)

wrData: Write data buffer

length: Number of bytes to be written to the slave

repeated: Repeated start

0: Stop bit is generated at the end

1: Stop bit is not generated at the end

```
3.14.4.    UINT8_T I2C0_WAIT_UNTIL_DONE(VOID)
            UINT8_T I2C1_WAIT_UNTIL_DONE(VOID)
```

Returns i2c status if I2C is done with the transfer else put chip into standby mode and wait for the interrupt to wake-up. This is a blocking function.

```
3.14.5.    UINT8_T I2C0_GET_STATUS(VOID)
            UINT8_T I2C1_GET_STATUS(VOID)
```

Returns I2C status

I2C_status can be any of the following.

AXM0_STATUS_SUCCESS

AXM0_STATUS_FAIL

AXM0_STATUS_INVALID_PARAMETER

LIBMF_I2C_BUS_ERROR

LIBMF_I2C_ARB_LOST

AXM0_STATUS_IN_PROGRESS

AXM0_STATUS_NO_INIT

```
3.14.6.    uint8_t I2C0_CLOSE(VOID)
           uint8_t I2C1_CLOSE(VOID)
```

Disables I2C and returns status.

Usage example 1: I2C write - blocking for the status

```
uint8_t i2c_status = i2c0_init(LIBMF_I2C_MASTER, LIBMF_I2C_STANDARD_MODE,
LIBMF_I2C_INT_DIV_0, LIBMF_I2C_INTERNAL_CLK);

If(i2c_status == AXM0_STATUS_SUCCESS)
{
    uint8_t wtbuff[] = {user data}; /* This buffer will be used for i2c data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    uint16_t Slave_address; /* Address of I2C Slave */

    uint8_t Repeated; /* 0 – Stop at end, 1 – No Stop at end */

    i2c0_write(Slave_address, wtbuff, data_len, Repeated);

    i2c_status = i2c0_wait_until_done();
    If(i2c_status == AXM0_STATUS_SUCCESS)
    {
        /* i2c finished with the transfer */
    }
}
```

Usage example 2: I2C read - blocking for the status

```
uint8_t i2c_status = i2c0_init(LIBMF_I2C_MASTER, LIBMF_I2C_STANDARD_MODE,
LIBMF_I2C_INT_DIV_0, LIBMF_I2C_INTERNAL_CLK);

If(i2c_status == AXM0_STATUS_SUCCESS)
{
    uint8_t rdbuf[n]; /* This buffer will be used for i2c data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    uint16_t Slave_address; /* Address of I2C Slave */

    uint8_t Repeated; /* 0 – Stop at end, 1 – No Stop at end */
```



```

i2c0_read(Slave_address, rdbuf, data_len, Repeated);

i2c_status = i2c0_wait_until_done();
If(i2c_status == AXM0_STATUS_SUCCESS)
{
    /* Read data will be available in rdbuf */
    /* i2c finished with the transfer */
}
}

```

Usage example 3: I2C write - non-blocking

```

uint8_t i2c_status = i2c0_init(LIBMF_I2C_MASTER, LIBMF_I2C_STANDARD_MODE,
LIBMF_I2C_INT_DIV_0, LIBMF_I2C_INTERNAL_CLK);

If(i2c_status == AXM0_STATUS_SUCCESS)
{
    uint8_t rdbuf[] = {user data}; /* This buffer will be used for i2c data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    uint16_t Slave_address; /* Address of I2C Slave */

    uint8_t Repeated; /* 0 – Stop at end, 1 – No Stop at end */

    i2c0_write(Slave_address, wrbuf, data_len, Repeated);

    while(i2c0_get_status() == AXM0_STATUS_IN_PROGRESS)
    {
        /* Do user tasks if any */
    }
    /* i2c finished with the transfer */
}
}

```

Usage example 4: I2C read - non-blocking

```

uint8_t i2c_status = i2c0_init(LIBMF_I2C_MASTER, LIBMF_I2C_STANDARD_MODE,
LIBMF_I2C_INT_DIV_0, LIBMF_I2C_INTERNAL_CLK);

If(i2c_status == AXM0_STATUS_SUCCESS)
{
    uint8_t rdbuf[] = {user data}; /* This buffer will be used for i2c data transfer
until finished, recommended to allocate in data segment */

    uint8_t data_len = user length;

    uint16_t Slave_address; /* Address of I2C Slave */

```

```

uint8_t Repeated; /* 0 – Stop at end, 1 – No Stop at end */

i2c0_read(Slave_address, rdbuf, data_len, Repeated);

while(i2c0_get_status() == AXM0_STATUS_IN_PROGRESS)
{
    /* Do user tasks if any */
}
/* Read data will be available in rdbuf */
/* i2c finished with the transfer */
}

```

3.15. ADC DRIVER

AXM0F243:

These driver APIs are implemented to support single ended and Differential voltage on supported GPIO pins, VDDIO voltage on supported GPIO pins and Internal die temperature measurement.

These APIs are exposed through 'libmfadc.h', Package pin details are exposed in 'axm0f2_pin.h'

3.15.1. UINT8_T ADC_INIT(UINT8_T POSITIVE_INPUT, UINT8_T
 NEGATIVE_INPUT, UINT8_T CONV_MODE, UINT8_T REF_INPUT, UINT16_T
 EXT_VREF)

This API initializes ADC for the below user specified configurations.

positive_input:

```

ADC_TEMPERATURE_MEASUREMENT (Only conv_mode LIBMF_ADC_ONESHOT and
ref_input LIBMF_ADC_VREF_INTERNAL is Supported)
ADC_VDDIO_MEASUREMENT (Only conv_mode LIBMF_ADC_ONESHOT and ref_input
LIBMF_ADC_VREF_INTERNAL is Supported)
PKG_PIN_NUM_01
PKG_PIN_NUM_02
.
.
.
PKG_PIN_NUM_40

```

negative_input:

```

PKG_PIN_GROUND
PKG_PIN_NUM_01
PKG_PIN_NUM_02
.
.
.
PKG_PIN_NUM_40

```

conv_mode:

LIBMF_ADC_CONTINUOUS
LIBMF_ADC_ONESHOT

ref_input:

LIBMF_ADC_VREF_INTERNAL : 1.2 V
LIBMF_ADC_VREF_VDDA_BY_2 : 1.65V
LIBMF_ADC_VREF_VDDA : 3.3V
LIBMF_ADC_VREF_EXTERNAL : Input on pin 1.7

ext_vref: This passing parameter is valid if *ref_input* is LIBMF_ADC_VREF_EXTERNAL, External reference input voltage in millivolts from 0 to 3300.

If 1.2V is supplied on Pin 1.7 for ref_input LIBMF_ADC_VREF_EXTERNAL, ext_vref is 1200

If 3.0V is supplied on Pin 1.7 for ref_input LIBMF_ADC_VREF_EXTERNAL, ext_vref is 3000

For ref_input other than LIBMF_ADC_VREF_EXTERNAL, ext_vref is 0.

Returns ADC status

3.15.2. VOID ADC_START_MEASUREMENT (VOID)

ADC will be started and measurement will be done in interrupt context. ADC status can be checked with `adc_wait_until_done()/adc_get_status()`. This API is required only for one shot conversion mode.

3.15.3. UINT8_T ADC_WAIT_UNTIL_DONE (VOID)

This function can be used to poll the ADC and to check whether ADC is done with the measurement. If ADC is busy, chip will enter standby mode and wakes-up upon ADC interrupts. This function returns ADC status on completion. This API is required only for one shot conversion mode. This is a blocking function.

3.15.4. UINT8_T ADC_GETSTATUS (VOID)

Returns ADC status.

ADC status can be any of the following

AXMO_STATUS_SUCCESS
AXMO_STATUS_FAIL
AXMO_STATUS_INVALID_PARAMETER
AXMO_STATUS_IN_PROGRESS
AXMO_STATUS_NO_INIT

3.15.5. UINT8_T ADC_GET_RESULT (VOID)

Returns measured ADC Result.

3.15.6. INT16_T ADC_RESULT_IN_MVOLTS(INT16_T ADCCOUNTS)

This function converts the ADC result into millivolts.

adcCounts: 11 Bit Signed ADC digital value

Returns voltage in millivolts(mV)

3.15.7. INT32_T ADC_CONVERT_DIETEMP(INT32_T ADCCOUNTS)

This function converts the ADC measured temperature data into degree Celsius.

adcCounts: 11 Bit Signed ADC digital value.

Returns Temperature in Celsius.

3.15.8. VOID ADC_CLOSE(VOID)

This function disables ADC module.

Usage example 1: Internal Die Temperature Measurement

```
uint8_t adc_status = adc_init(ADC_TEMPERATURE_MEASUREMENT,PKG_PIN_GROUND,
LIBMF_ADC_ONESHOT, LIBMF_ADC_VREF_INTERNAL,0);

if(adc_status == AXM0_STATUS_SUCCESS)
{
    adc_start_measurement();

    adc_status = adc_wait_until_done();
    if(adc_status == AXM0_STATUS_SUCCESS)
    {
        int16_t adcResult = adc_get_result(); /* adcResult will have 11 bit ADC value */

        int32_t adcResult_temp = adc_convert_dietemp(adcResult);
        /* adcResult_temp will have temperature value in hex. If adcResult_temp = 0x2001,
        Temperature is 32.01 deg celsius */
    }
}

adc_close();
```

Usage example 2: Continuous Single ended Voltage Measurement

```
uint8_t adc_status = adc_init(PKG_PIN_NUM_18,PKG_PIN_GROUND,
LIBMF_ADC_CONTINUOUS, LIBMF_ADC_VREF_INTERNAL,0);

if(adc_status == AXM0_STATUS_SUCCESS)
{
```

```
int16_t adcResult = adc_get_result(); /* adcResult will have 11 bit ADC value */

int16_t voltage = adc_result_in_mVolts(adcResult);

/* voltage will have output voltage in millivolts */
}

adc_close();
```

Usage example 3: One Shot Single ended Voltage Measurement

```
uint8_t adc_status = adc_init(PKG_PIN_NUM_18,PKG_PIN_GROUND, LIBMF_ADC_ONESHOT,
LIBMF_ADC_VREF_INTERNAL,0);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    adc_start_measurement();

    adc_status = adc_wait_until_done();

    if(adc_status == AXMO_STATUS_SUCCESS)
    {
        int16_t adcResult = adc_get_result(); /* adcResult will have 11 bit ADC value */

        int16_t voltage = adc_result_in_mVolts(adcResult);

        /* voltage will have output voltage in millivolts */
    }
}

adc_close();
```

Usage example 4: Continuous Differential Voltage Measurement

```
uint8_t adc_status = adc_init(PKG_PIN_NUM_18, PKG_PIN_NUM_31,
LIBMF_ADC_CONTINUOUS, LIBMF_ADC_VREF_INTERNAL,0);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    int16_t adcResult = adc_get_result(); /* adcResult will have 11 bit ADC value */

    int16_t voltage = adc_result_in_mVolts(adcResult);

    /* voltage will have output voltage in millivolts */
}

adc_close();
```

Usage example 5: One Shot Differential Voltage Measurement

```
uint8_t adc_status = adc_init(PKG_PIN_NUM_18, PKG_PIN_NUM_31, LIBMF_ADC_ONESHOT,
LIBMF_ADC_VREF_INTERNAL,0);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    adc_start_measurement();

    adc_status = adc_wait_until_done();

    if(adc_status == AXMO_STATUS_SUCCESS)
    {

        int16_t adcResult = adc_get_result();  /* adcResult will have 11 bit ADC value */

        int16_t voltage = adc_result_in_mVolts(adcResult);

        /* voltage will have output voltage in millivolts */
    }
}

adc_close();
```

Usage example 6: Continuous Single ended Voltage Measurement, External Voltage reference, 1.2V supplied on Pin 1.7

```
uint8_t adc_status = adc_init(PKG_PIN_NUM_18,PKG_PIN_GROUND,
LIBMF_ADC_CONTINUOUS, LIBMF_ADC_VREF_EXTERNAL,1200);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    int16_t adcResult = adc_get_result();  /* adcResult will have 11 bit ADC value */

    int16_t voltage = adc_result_in_mVolts(adcResult);

    /* voltage will have output voltage in millivolts */
}

adc_close();
```

Usage example 7: One Shot Differential Voltage Measurement, External Voltage reference, 3V supplied on Pin 1.7

```
uint8_t adc_status = adc_init(PKG_PIN_NUM_18, PKG_PIN_NUM_31, LIBMF_ADC_ONESHOT,
LIBMF_ADC_VREF_EXTERNAL,3000);

if(adc_status == AXMO_STATUS_SUCCESS)
{
```

```
    adc_start_measurement();

    adc_status = adc_wait_until_done();

    if(adc_status == AXMO_STATUS_SUCCESS)
    {

        int16_t adcResult = adc_get_result();    /* adcResult will have 11 bit ADC value */

        int16_t voltage = adc_result_in_mVolts(adcResult);

        /* voltage will have output voltage in millivolts */
    }
}

adc_close();
```

Usage example 8: VDDIO Measurement on Pin 3.6

```
uint8_t adc_status = adc_init(ADC_VDDIO_MEASUREMENT, PKG_PIN_NUM_16,
LIBMF_ADC_ONESHOT, LIBMF_ADC_VREF_INTERNAL,0);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    adc_start_measurement();

    adc_status = adc_wait_until_done();

    if(adc_status == AXMO_STATUS_SUCCESS)
    {

        int16_t adcResult = adc_get_result();    /* adcResult will have VDDIO value */

        int16_t voltage = adc_result_in_mVolts(adcResult);

        /* voltage will have output VDDIO voltage in millivolts */
    }
}

adc_close();
```

3.16. PWM DRIVER

AXM0F243:

- PWM driver APIs are implemented to support PWM functionality of TCPWM Module.
- There are 5 TCPWM - TCPWM0, TCPWM1, TCPWM2, TCPWM3, TCPWM4.
- PWM APIs Support TCPWM1, TCPWM2, TCPWM3.
- TCPWM0 and TCPWM4 are not supported due to pin restrictions.
- PWM Driver supports Left Aligned, Right Aligned, Center Aligned and Assymetric Aligned Modes.
- In Left and Right Aligned Modes, PWM Period supported is 3 Microsec to 174.5 milli sec.
- In Center and Assymetric Aligned Modes, PWM Period supported is 6 Microsec to 349 milli sec.
- Duty Cycle is supported in terms of percentage 0 to 100.
- Integer clock dividers supported from 0 to 5.

3.16.1. UINT8_T PWM_INIT(UINT8_T SEL_PWM ,UINT8_T PWM_PIN, UINT8_T PWM_TYPE, UINT32_T PWM_PERIOD, UINT8_T PWM_DUTYCYCLE, UINT8_T SEL_DIV)

This function shall be used to initialize PWM for user specified parameters of PWM, Select PWM TCPWM1 or TCPWM2 or TCPWM3, Line Out Pin For Selected TCPWM, PWM Type Supported are Left Aligned, Right Aligned, Center Aligned, Assymetric Aligned, PWM Period in Micro Seconds, PWM Duty Cycle in percentage 0 to 100.

Input:

sel_pwm : Select the TCPWM for operation
LIBMF_PWM_1 (TCPWM1)
LIBMF_PWM_2 (TCPWM2)
LIBMF_PWM_3 (TCPWM3)

pwm_pin : Select the Pin on which PWM activity is seen.
PKG_PIN_NUM_14 (P3.2) for LIBMF_PWM_1
PKG_PIN_NUM_30 (P1.0) for LIBMF_PWM_2
PKG_PIN_NUM_32 (P1.2) for LIBMF_PWM_3
PKG_PIN_NUM_16 (P3.6) for LIBMF_PWM_3

pwm_type : Select the PWM type
LIBMF_PWM_LEFT_ALIGNED
LIBMF_PWM_RIGHT_ALIGNED
LIBMF_PWM_CENTER_ALIGNED
LIBMF_PWM_ASSYMETRIC_ALIGNED

pwm_period : Period in Micro Seconds

3 to 174500 Microsec in LIBMF_PWM_LEFT_ALIGNED and LIBMF_PWM_RIGHT_ALIGNED

6 to 349000 Microsec in LIBMF_PWM_CENTER_ALIGNED and
LIBMF_PWM_ASSYMETRIC_ALIGNED

pwm_dutycycle : Percentage of pwm_period (0 to 100)

sel_div : Integer Clock Divider selection (0 to 5)

Return : Returns status

AXMO_STATUS_SUCCESS
AXMO_STATUS_INVALID_PARAMETER

3.16.2. UINT8_T PWM_DUTYCYCLE (UINT8_T SEL_PWM, UINT8_T
PWM_DUTYCYCLE)

This function changes duty cycle of pulse, in percentage of pwm period.

Input:

sel_pwm : Select the TCPWM for operation
LIBMF_PWM_1 (TCPWM1)
LIBMF_PWM_2 (TCPWM2)
LIBMF_PWM_3 (TCPWM3)

pwm_dutycycle : Percentage of Period (0 to 100)

Return : Returns status

Status : AXMO_STATUS_SUCCESS
AXMO_STATUS_NO_INIT
AXMO_STATUS_INVALID_PARAMETER

3.16.3. UINT8_T PWM_START (UINT8_T SEL_PWM)

This function enables and starts the counter for PWM operation

Input:

sel_pwm : Select the TCPWM for operation
LIBMF_PWM_1 (TCPWM1)
LIBMF_PWM_2 (TCPWM2)
LIBMF_PWM_3 (TCPWM3)

Return : Returns status

Status : AXMO_STATUS_SUCCESS
AXMO_STATUS_NO_INIT
AXMO_STATUS_INVALID_PARAMETER

3.16.4. `UINT8_T PWM_STOP (UINT8_T SEL_PWM)`

This function stops the counter for PWM operation

Input:

sel_pwm : Select the TCPWM for operation
 LIBMF_PWM_1 (TCPWM1)
 LIBMF_PWM_2 (TCPWM2)
 LIBMF_PWM_3 (TCPWM3)

Return : Returns status

Status : AXMO_STATUS_SUCCESS
 AXMO_STATUS_NO_INIT
 AXMO_STATUS_INVALID_PARAMETER

3.16.5. `UINT8_T PWM_CLOSE (UINT8_T SEL_PWM)`

This function stops and disables the counter for PWM operation

Input:

sel_pwm : Select the TCPWM for operation
 LIBMF_PWM_1 (TCPWM1)
 LIBMF_PWM_2 (TCPWM2)
 LIBMF_PWM_3 (TCPWM3)

Return : Returns status

Status : AXMO_STATUS_SUCCESS
 AXMO_STATUS_NO_INIT
 AXMO_STATUS_INVALID_PARAMETER

NOTE:

1. After PWM Init and PWM Start, if the application puts the controller to deepsleep, After controller wakeup PWM start has to be done to start PWM activity on selected pin.

Usage example 1: PWM Pulse with period 1milli sec, Center aligned mode, 50% dutycycle on pin 3.6 using TCPWM3, Integer divider 1 is used for Clock division.

```
uint8_t adc_status = pwm_init (LIBMF_PWM_3,PKG_PIN_NUM_16,LIBMF_PWM_CEN-
TER_ALIGNED, 1000,50,1);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    pwm_start(LIBMF_PWM_3);
}

/* If Dutycycle of PWM needs to be changed to 75% */
pwm_dutycycle(LIBMF_PWM_3,75);
```

```
/* If PWM pulse needs to be stopped */
pwm_stop(LIBMF_PWM_3);

/* If PWM needs to be disabled */
pwm_close(LIBMF_PWM_3);
```

Usage example 2: PWM Pulse with period 6 Micro Sec, Center aligned mode, 25% dutycycle on pin 3.2 using TCPWM1, Integer divider 5 is used for Clock division.

```
uint8_t adc_status = pwm_init (LIBMF_PWM_1,PKG_PIN_NUM_14,LIBMF_PWM_CEN-
TER_ALIGNED, 6,25,5);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    pwm_start(LIBMF_PWM_1);
}

/* If Dutycycle of PWM needs to be changed to 75% */
pwm_dutyicycle(LIBMF_PWM_1,75);

/* If PWM pulse needs to be stopped */
pwm_stop(LIBMF_PWM_1);

/* If PWM needs to be disabled */
pwm_close(LIBMF_PWM_1);
```

Usage example 3: PWM Pulse with period 349 milli Sec, Asymmetric aligned mode, 10% dutycycle on pin 1.0 using TCPWM2, Integer divider 0 is used for Clock division.

```
uint8_t adc_status = pwm_init (LIBMF_PWM_2,PKG_PIN_NUM_30, LIBMF_PWM_ASSYMET-
RIC_ALIGNED, 349000,10,0);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    pwm_start(LIBMF_PWM_2);
}

/* If Dutycycle of PWM needs to be changed to 40% */
pwm_dutyicycle(LIBMF_PWM_2,40);

/* If PWM pulse needs to be stopped */
pwm_stop(LIBMF_PWM_2);

/* If PWM needs to be disabled */
```

```
pwm_close(LIBMF_PWM_2);
```

Usage example 4: PWM Pulse with period 3 Micro Sec, Left aligned mode, 70% dutycycle on pin 1.2 using TCPWM3, Integer divider 2 is used for Clock division.

```
uint8_t adc_status = pwm_init (LIBMF_PWM_3,PKG_PIN_NUM_32,
LIBMF_PWM_LEFT_ALIGNED, 3,70,2);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    pwm_start(LIBMF_PWM_3);
}

/* If Dutycycle of PWM needs to be changed to 25% */
pwm_dutycycle(LIBMF_PWM_3,25);

/* If PWM pulse needs to be stopped */
pwm_stop(LIBMF_PWM_3);

/* If PWM needs to be disabled */
pwm_close(LIBMF_PWM_3);
```

Usage example 5: PWM Pulse with period 174.5 milli Sec, Right aligned mode, 60% dutycycle on pin 3.6 using TCPWM3, Integer divider 4 is used for Clock division.

```
uint8_t adc_status = pwm_init (LIBMF_PWM_3,PKG_PIN_NUM_16,
LIBMF_PWM_RIGHT_ALIGNED, 174500,60,4);

if(adc_status == AXMO_STATUS_SUCCESS)
{
    pwm_start(LIBMF_PWM_3);
}

/* If Dutycycle of PWM needs to be changed to 50% */
pwm_dutycycle(LIBMF_PWM_3,50);

/* If PWM pulse needs to be stopped */
pwm_stop(LIBMF_PWM_3);

/* If PWM needs to be disabled */
pwm_close(LIBMF_PWM_3);
```

4. CONVENIENCE FUNCTIONS

4.1. LIBMFBCH.H

Bose-Chaudhuri-Hocquenghem (BCH) codes are a class of cyclic error correcting codes. Error correcting codes add redundancy to code words, which allows the receiver to correct some bit errors.

This header file provides functions which work on BCH (31, 21, and 5) code words, with an additional parity bit. This code is primarily used in paging systems. This code can correct up to two erroneous bits per 32 bit code word.

The routines work on 32 bit words. Bits 31—11 are the 21 data bits. Bits 10—1 are the BCH parity bits. Bit 0 is an even parity bit.

4.1.1. `UINT16_T BCH3121_SYNDROME(UINT32_T CW)`

This function computes the syndrome for the code word. Bit 0 needs not be valid.

4.1.2. `UINT32_T BCH3121_ENCODE(UINT32_T CW)`

This function calculates the BCH parity bits. The argument only needs to have the data part valid. It returns the data with the BCH parity bits added. The additional parity bit is not computed.

4.1.3. `UINT32_T BCH3121_ENCODE_PARITY(UINT32_T CW)`

This function performs the same calculation as `bch3121_encode`, and additionally also computes the additional parity bit.

4.1.4. `UINT32_T BCH3121_DECODE(UINT32_T CW)`

This function tries to decode a BCH code word. It returns the code word with bit errors corrected and bit 0 cleared if successful, or bit 1 set to indicate a decoding error. A decoding error can happen if there were more than 2 bit errors in the code word.

4.1.5. `UINT32_T BCH3121_DECODE_PARITY(UINT32_T CW)`

This function performs the same calculation as `bch3121_decode`, but additionally checks whether the additional parity bit matches the (corrected) code word parity.

4.2. LIBMFCRC.H

4.2.1. `UINT8_T CRC8_CCITT_BYTE(UINT8_T CRC, UINT8_T C)`

This function computes an 8-bit CRC (Cyclic Redundancy Check) (polynomial $x^8+x^2+x^1+1$) of one data byte `c` given the initial state vector `CRC`. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This routine does not use a table, and is therefore smaller but slower than the table driven version below.

4.2.2. `UINT8_T CRC8_CCITT(CONST UINT8_T *BUF, UINT8_T LEN, UINT8_T INIT)`

This function computes an 8-bit CRC (Cyclic Redundancy Check) (polynomial $x^8+x^2+x^1+1$) of a buffer addressed by buf of length len with the initial state vector init. This routine does not use a table, and is therefore smaller but slower than the table driven version below.

4.2.3. `UINT8_T CRC8_ONEWIRE_BYTE(UINT8_T CRC, UINT8_T C)`

This function computes an 8-bit CRC (Cyclic Redundancy Check) (polynomial $x^8+x^5+x^4+1$) of one data byte c given the initial state vector crc. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This routine does not use a table, and is therefore smaller but slower than the table driven version below.

4.2.4. `UINT8_T CRC8_ONEWIRE(CONST UINT8_T *BUF, UINT8_T LEN, UINT8_T INIT)`

This function computes an 8-bit CRC (Cyclic Redundancy Check) (polynomial $x^8+x^5+x^4+1$) of a buffer addressed by buf of length len with the initial state vector init. This routine does not use a table, and is therefore smaller but slower than the table driven version below.

4.2.5. `UINT8_T CRC_CRC8CCITT_BYTE(UINT8_T CRC, UINT8_T C)`

This function computes the 8-bit CCITT CRC (Cyclic Redundancy Check) (polynomial $x^8+x^2+x^1+1$) of one data byte c given the initial state vector crc. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c LSB first, and the state vector is bit reversed.

4.2.6. `UINT8_T CRC_CRC8CCITT_MSB_BYTE(UINT8_T CRC, UINT8_T C)`

This function computes the 8-bit CCITT CRC (Cyclic Redundancy Check) (polynomial $x^8+x^2+x^1+1$) of one data byte c given the initial state vector crc. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c MSB first.

4.2.7. `UINT8_T CRC_CRC8ONEWIRE_BYTE(UINT8_T CRC, UINT8_T C)`

This function computes the 8-bit OneWire CRC (Cyclic Redundancy Check) (polynomial $x^8+x^5+x^4+1$) of one data byte c given the initial state vector crc. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c LSB first, and the state vector is bit reversed.

4.2.8. `UINT8_T CRC_CRC8ONEWIRE_MSB_BYTE(UINT8_T CRC, UINT8_T C)`

This function computes the 8-bit OneWire CRC (Cyclic Redundancy Check) (polynomial $x^8+x^5+x^4+1$) of one data byte c given the initial state vector CRC. The CRC of a whole buffer

may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c MSB first.

4.2.9. `UINT16_T CRC_CCITT_BYTE(UINT16_T CRC, UINT8_T C)`

This function computes the 16-bit CCITT CRC (Cyclic Redundancy Check) (polynomial $x^{16}+x^{12}+x^5+1$) of one data byte c given the initial state vector crc. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c LSB first, and the state vector is bit reversed.

4.2.10. `UINT16_T CRC_CCITT_MSB_BYTE(UINT16_T CRC, UINT8_T C)`

This function computes the 16-bit CCITT CRC (Cyclic Redundancy Check) (polynomial $x^{16}+x^{12}+x^5+1$) of one data byte c given the initial state vector CRC. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c MSB first.

4.2.11. `UINT16_T CRC_CRC16_BYTE(UINT16_T CRC, UINT8_T C)`

This function computes the 16-bit CRC-16 CRC (Cyclic Redundancy Check) (polynomial $x^{16}+x^{15}+x^2+1$) of one data byte c given the initial state vector CRC. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c LSB first, and the state vector is bit reversed.

4.2.12. `UINT16_T CRC_CRC16_MSB_BYTE(UINT16_T CRC, UINT8_T C)`

This function computes the 16-bit CRC-16 CRC (Cyclic Redundancy Check) (polynomial $x^{16}+x^{15}+x^2+1$) of one data byte c given the initial state vector CRC. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c MSB first.

4.2.13. `UINT16_T CRC_CRC16DNP_BYTE(UINT16_T CRC, UINT8_T C)`

This function computes the 16-bit CRC-16 DNP CRC (Cyclic Redundancy Check) (polynomial $x^{16}+x^{13}+x^{12}+x^{11}+x^{10}+x^8+x^6+x^5+x^2+1$) of one data byte c given the initial state vector CRC. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c LSB first, and the state vector is bit reversed.

4.2.14. `UINT16_T CRC_CRC16DNP_MSB_BYTE(UINT16_T CRC, UINT8_T C)`

This function computes the 16-bit CRC-16 DNP CRC (Cyclic Redundancy Check) (polynomial $x^{16}+x^{13}+x^{12}+x^{11}+x^{10}+x^8+x^6+x^5+x^2+1$) of one data byte c given the initial state vector crc. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes c MSB first.

4.2.15. `UINT32_T CRC_CRC32_BYTE(UINT32_T CRC, UINT8_T C)`

This function computes the 32-bit CRC-32 CRC (Cyclic Redundancy Check) (polynomial $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$) of one data byte `c` given the initial state vector `crc`. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes `c` LSB first, and the state vector is bit reversed.

4.2.16. `UINT32_T CRC_CRC32_MSB_BYTE(UINT32_T CRC, UINT8_T C)`

This function computes the 32-bit CRC-32 CRC (Cyclic Redundancy Check) (polynomial $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x^1+1$) of one data byte `c` given the initial state vector `crc`. The CRC of a whole buffer may be computed by calling this function repeatedly for each byte, supplying the return value of the previous byte as state vector input of the current byte. This function processes `c` MSB first.

4.2.17. `UINT8_T CRC_CRC8CCITT(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT8_T CRC)` `UINT8_T CRC_CRC8CCITT_MSB(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT8_T CRC)` `UINT8_T CRC_CRC8ONEWIRE(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT8_T CRC)` `UINT8_T CRC_CRC8ONEWIRE_MSB(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT8_T CRC)` `UINT16_T CRC_CCITT(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT16_T CRC)` `UINT16_T CRC_CCITT_MSB(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT16_T CRC)` `UINT16_T CRC_CRC16(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT16_T CRC)` `UINT16_T CRC_CRC16_MSB(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT16_T CRC)` `UINT16_T CRC_CRC16DNP(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT16_T CRC)` `UINT16_T CRC_CRC16DNP_MSB(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT16_T CRC)` `UINT32_T CRC_CRC32(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT32_T CRC)` `UINT32_T CRC_CRC32_MSB(CONST UINT8_T *BUF, UINT16_T BUFLen, UINT32_T CRC)`

These functions compute the CRC of a full buffer. The initial CRC register value is passed in parameter `crc`, the final `crc` value is returned. The buffer length may be zero, in which case the initial CRC value is returned. See a description of the bitwise CRC functions above for a description of the polynomials and the bit order.

4.2.18. UINT16_T PN9_ADVANCE(UINT16_T PN9)

This function computes the next state value (i.e. after one byte that is after eight bits) of the PN-9 whitening sequence. The polynomial of the PN-9 sequence is x^9+x^5+1 . This function uses a table-driven implementation, requiring 512 bytes of table data.

4.2.19. UINT16_T PN9_ADVANCE_BIT(UINT16_T PN9)

This function computes the PN-9 whitening sequence state after one bit.

4.2.20. UINT16_T PN9_ADVANCE_BITS(UINT16_T PN9, UINT16_T BITS)

This function computes the PN-9 whitening sequence state after a given number of bits.

4.2.21. UINT16_T PN9_ADVANCE_BYTE(UINT16_T PN9)

This function computes the PN-9 whitening sequence state after a byte (that is, eight bits). It computes the same value as `pn9_advance`, but uses a bit-wise implementation instead of a table-driven implementation. It therefore requires less FLASH space, but may take slightly longer, depending on compiler optimization.

4.2.22. UINT16_T PN9_BUFFER(UINT8_T __GENERIC *BUF, UINT16_T BUFLen, UINT16_T PN9, UINT8_T XOR)

This function xor's a buffer with a PN-9 sequence and the additional xor byte `xor`. The PN-9 start value is given by the parameter `pn9`, and the final value is returned. The additional xor byte may be used to invert the buffer by setting it to 0xff.

5. HISTORY


Version	Date	Comments
1.13		Added support for AX8052/AX8052F143 library
1.14	28-Aug-18	Added support for AXM0F243 library
1.15	24-Jan-19	Added calibration details
1.16	7-Feb-19	Updated calibration details and format changes.
1.17	10-May-19	Added interface functions for ADC, SPI and I2C drivers.
1.18	11-Sept-19	Added interface functions for PWM
1.19	18-Nov-19	Added APIs for AX5044_45

6. 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.

ON Semiconductor and  are trademarks of Semiconductor Components Industries, LLC dba ON Semiconductor or its subsidiaries in the United States and/or other countries. ON Semiconductor owns the rights to a number of patents, trademarks, copyrights, trade secrets, and other intellectual property. A listing of ON Semiconductor's product/patent coverage may be accessed at www.onsemi.com/site/pdf/Patent-Marketing.pdf. ON Semiconductor reserves the right to make changes without further notice to any products herein. ON Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does ON Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation special, consequential or incidental damages. Buyer is responsible for its products and applications using ON Semiconductor products, including compliance with all laws, regulations and safety requirements or standards, regardless of any support or applications information provided by ON Semiconductor. "Typical" parameters which may be provided in ON Semiconductor datasheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. ON Semiconductor does not convey any license under its patent rights nor the rights of others. ON Semiconductor products are not designed, intended, or authorized for use as a critical component in life support systems or any FDA Class 3 medical devices or medical devices with a same or similar classification in a foreign jurisdiction or any devices intended for implantation in the human body. Should Buyer purchase or use ON Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold ON Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that ON Semiconductor was negligent regarding the design or manufacture of the part. ON Semiconductor is an Equal Opportunity/Affirmative Action Employer. This literature is subject to all applicable copyright laws and is not for resale in any manner.