Software Manual

LibMF (AX8052 Support Library)

Version 1.13

AXSEM

Table of Contents

1. Introduction

LibMF is a convenience library to ease the use of Axsem AX8052 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.

2. Microprocessor Functions

2.1. ax8052.h, ax8052f131.h, ax8052f142.h, ax8052f143.h, ax8052f151.h

These headers define the special function registers (SFR) of the Axsem AX8052 Microprocessor. `ax8052.h` defines the SFR's of the microcontroller core only. It is suitable for the AX8052F101, or the AX8052F100 without any Axsem radio chip connected. `ax8052f131.h`, `ax8052f142.h`, `ax8052f143.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.

2.2. 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.

2.3. LIBMFTYPES.H

2.3.1. 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.

2.3.2. __CODE, __DATA, __XDATA, __PDATA, __GENERIC

These defines provide compiler independent address space specifiers.

2.3.3. 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.

2.3.4. 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.

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.

2.3.5. 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.

2.3.6. 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.

2.3.7. 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.

2.3.8. 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.

2.3.9. 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.

2.3.10. 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.

2.3.11. 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.

2.3.12. 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.

2.3.13. void fmemcpy(void __generic *d, const void __generic *s, uint16_t n)

fmemcpy 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.

2.3.14. 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 or IAR, it calls enter_standby(), below.

2.3.15. void enter_standby(void)

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.

2.3.16. void enter_sleep(void)

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.

2.3.17. void enter_sleep_cont(void)

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.

2.3.18. void enter_deepsleep(void)

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.

2.3.19. 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.

2.4. LIBMFDBGLINK.H

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 Axsem AxCode::Blocks IDE.

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

2.4.1. VOID DBGLINK_INIT(VOID)

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

2.4.2. 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.

2.4.3. 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.

2.4.4. 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.

2.4.5. 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 2.3.19

2.4.6. 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 2.3.19

2.4.7. uint8_t dbglink_txbuffersize(void)
uint8_t dbglink_rxbuffersize(void)

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

2.4.8. uint8_t dbglink_rxcount(void)

dbglink_rxcount returns the number of characters in the receive FIFO.

2.4.9. uint8_t dbglink_txfree(void)

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

2.4.10. 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.

2.4.11. void dbglink_wait_rxcount(uint8_t v)

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

2.4.12. void dbglink_wait_txdone(void)

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

2.4.13. 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.

2.4.14. void dbglink_rxadvance(uint8_t idx)

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

2.4.15. 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.

2.4.16. uint8_t dbglink_rxpeek(uint8_t idx)

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

2.4.17. 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.

2.4.18. 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.

2.4.19. 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.

2.4.20. DBGLINK_DEFINE_TXBUFFER(sz) DBGLINK_DEFINE_RXBUFFER(sz)

These macros define the 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_txbuffersize and dbglink_rxbuffersize will return one byte less than the argument.

2.5. libmfuart.h

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

2.5.1. 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.

2.6. libmfuart0.h, libmfuart1.h

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

2.6.1. 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 each direction and UART). It must be called before using the UART, and should be called before interrupts are globally enabled.

timernr (0...2) specifies the timer that should be used as baud rate generator. The corresponding timer must be initialized with uart_timer*x*_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.

2.6.2. 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.

2.6.3. 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.

2.6.4. 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.

2.6.5. 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.

2.6.6. 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 2.3.19

2.6.7. 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 2.3.19

2.6.8. uint8_t uart0_txbuffersize(void)
uint8_t uart1_txbuffersize(void)
uint8_t uart0_rxbuffersize(void)
uint8_t uart1_rxbuffersize(void)

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

2.6.9. uint8_t uart0_rxcount(void)
uint8_t uart1_rxcount(void)

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

2.6.10. 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.

2.6.11. 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.

2.6.12. 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.

2.6.13. 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.

2.6.14. 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.

2.6.15. 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.

2.6.16. 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.

2.6.17. 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.

2.6.18. 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.

2.6.19. 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.

2.6.20. 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.

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

These macros define the 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_txbuffersize / uart1_txbuffersize and uart0_rxbuffersize / uart1_rxbuffersize will return one byte less than the argument.

2.7. libmfflash.h

2.7.1. void flash_unlock(void)

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

2.7.2. 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.

2.7.3. 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 abritrary 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.

2.7.4. int8_t flash_write(uint16_t waddr, uint16_t wdata)

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.

2.7.5. 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 *.

2.7.6. uint8_t flash_apply_calibration(void)

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.

2.8. LIBMFRADIO.H

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

2.8.1. 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 16-bit, 24-bit or 32-bit (plus address) SPI transactions.

2.8.2. 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 16-bit, 24-bit or 32-bit (plus address) SPI transactions.

2.8.3. UINT8_T AX5031_RESET(VOID)
UINT8_T AX5042_RESET(VOID)
UINT8_T AX5043_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 plus the respective Axsem Radio Chip. There are small differences between the SoC and the two chip solution; these routines automatically detect and handle those differences.

2.8.4. VOID AX5031_COMMSLEEPEXIT(VOID)
VOID AX5042_COMMSLEEPEXIT(VOID)
VOID AX5043_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.

2.8.5. void ax5031_comminit(void)
void ax5042_comminit(void)
void ax5043_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.

2.8.6. void ax5031_rclk_enable(uint8_t div)
void ax5042_rclk_enable(uint8_t div)
void ax5043_rclk_enable(uint8_t div)
void ax5051_rclk_enable(uint8_t div)

These functions enable the radio system clock RSYSCLK. 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 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, RSYSCLK should not be enabled needlessly to save energy.

2.8.7. void ax5042_rclk_disable(void)
void ax5031_rclk_disable(void)
void ax5043_rclk_disable(void)
void ax5051_rclk_disable(void)

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

2.8.8. void ax5043_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.

2.8.9. uint8_t ax5043_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.

2.8.10. void ax5043_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.

2.8.11. void AX5043_WRITEFIFO(const uint8_t *PTR, uint8_t LEN)

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

2.9. libmfadc.h

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

2.9.1. 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.

2.9.2. uint16_t adc_singleended_offset_x1(void)

In single ended ×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 ×1 mode results. The return value of this function is only valid if called after flash_apply_calibration() has been called.

2.10. 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.

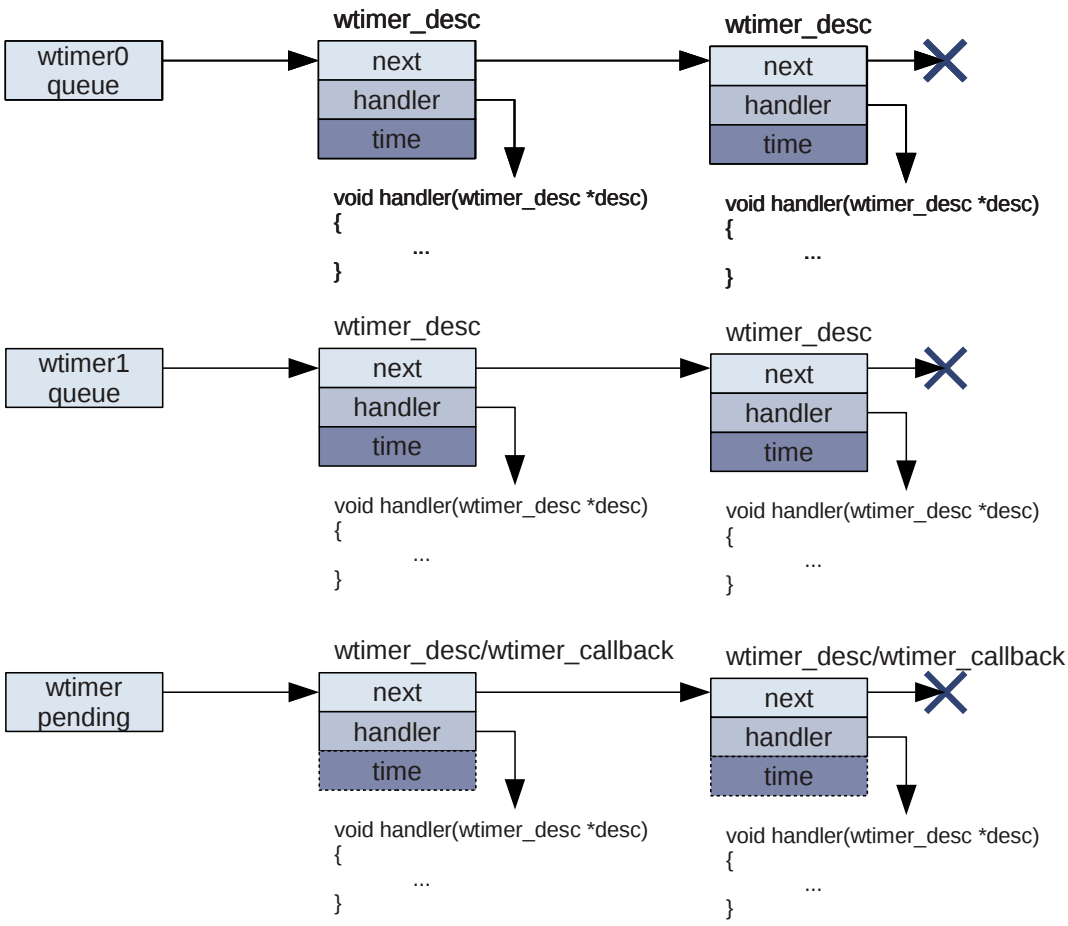


Figure 1: Wakeup Timer Queues

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, libmfwtimer.h must be included in the file containing the definition for main.

2.10.1. UINT32_T WTIMER0_CURTIME(VOID)
UINT32_T WTIMER1_CURTIME(VOID)

These functions retrieve the current time of the respective timer.

2.10.2. 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).

2.10.3. 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.

2.10.4. 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.

2.10.5. 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.

2.10.6. 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.

2.10.7. uint8_t wtimer_runcallbacks(void)

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

2.10.8. 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 version also returns zero if any of the used UARTs or DebugLink (if used) still has transmit data queued.

2.10.9. void wtimer0_setclksrc(uint8_t clksrc, unit8_t prescaler)
void wtimer1_setclksrc(uint8_t clksrc, unit8_t prescaler)

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 ×2, 1 means ×1, 2 means ÷2, 3 means ÷4, ... and 7 means ÷64.

2.10.10. void wtimer_init(void)

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

2.10.11. 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.

2.11. libmfosc.h

libmfosc.h contains oscillator related utility functions.

2.11.1. 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.

2.11.2. 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.

2.11.3. 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.

2.11.4. 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.

2.11.5. uint8_t setup_osc_calibration(uint32_t reffreq, uint8_t refosc)

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 Axsem Transceivers with a faster crystal or TCXO (such as the AX8052F143 with an 48MHz TCXO), it is recommended to divide SYSCLK in the transceiver block to obtain a reference clock within range (for example with the AX5043_PINFUNCSYSCLK register). This function returns zero on success, or non-zero if the input parameters are out of range.

2.11.6. setup_osc_calibration_const(reffreq,refosc)

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. Convenience Functions

3.1. libmfbch.h

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

This header file provides functions which work on BCH(31,21,5) codewords, 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 codeword.

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.

3.1.1. uint16_t bch3121_syndrome(uint32_t cw)

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

3.1.2. 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.

3.1.3. 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.

3.1.4. uint32_t bch3121_decode(uint32_t cw)

This function tries to decode a BCH codeword. It returns the codeword 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 codeword.

3.1.5. 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) codeword parity.

3.2. LIBMFCRC.H

3.2.1. 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.

3.2.2. 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.

3.2.3. 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.

3.2.4. 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.

3.2.5. 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.

3.2.6. 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.

3.2.7. 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.

3.2.8. 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.

3.2.9. 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.

3.2.10. 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.

3.2.11. 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.

3.2.12. 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.

3.2.13. 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.

3.2.14. 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.

3.2.15. 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.

3.2.16. 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.

3.2.17. 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 bytewise CRC functions above for a description of the polynomials and the bit order.

3.2.18. 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^4+1. This function uses a table-driven implementation, requiring 512 bytes of table data.

3.2.19. uint16_t pn9_advance_bit(uint16_t pn9)

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

3.2.20. 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.

3.2.21. 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.

3.2.22. uint16_t pn9_buffer(uint8_t __generic *buf, uint16_t buflen, uint16_t pn9, uint8_t xor)

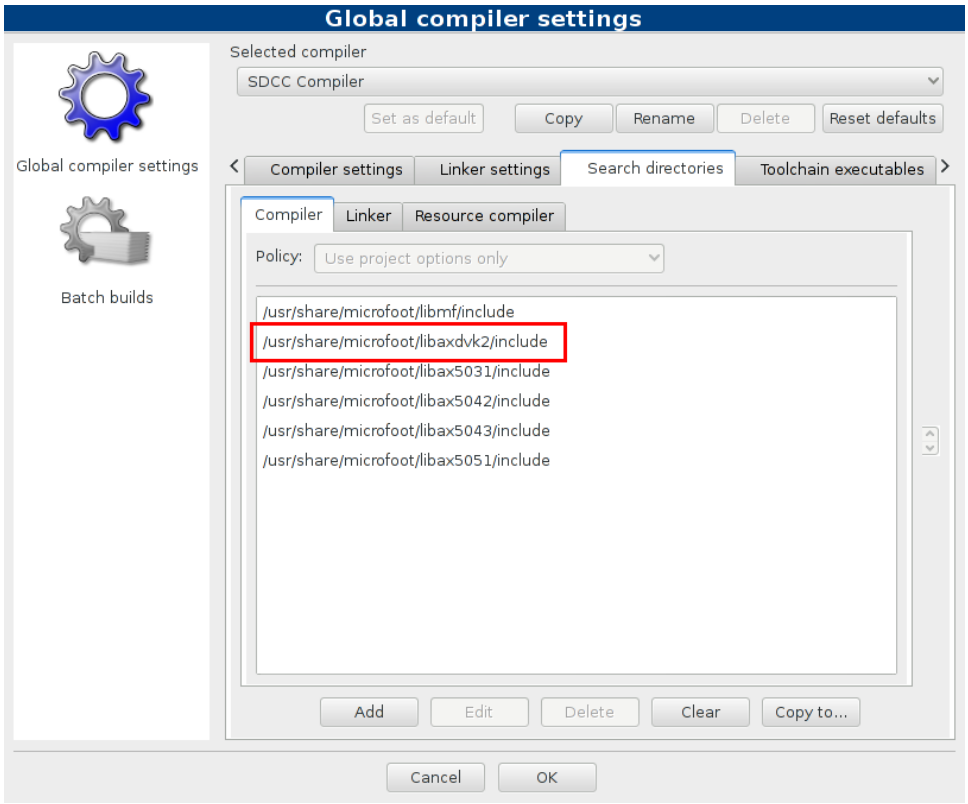
This function xor's a buffer with a PN-9 sequence and the additional xor byte xor. The PN-9 start value is given by the parameter pn9, and the final value is returned. The additional xor byte may be used to invert the buffer by setting it to 0xff.

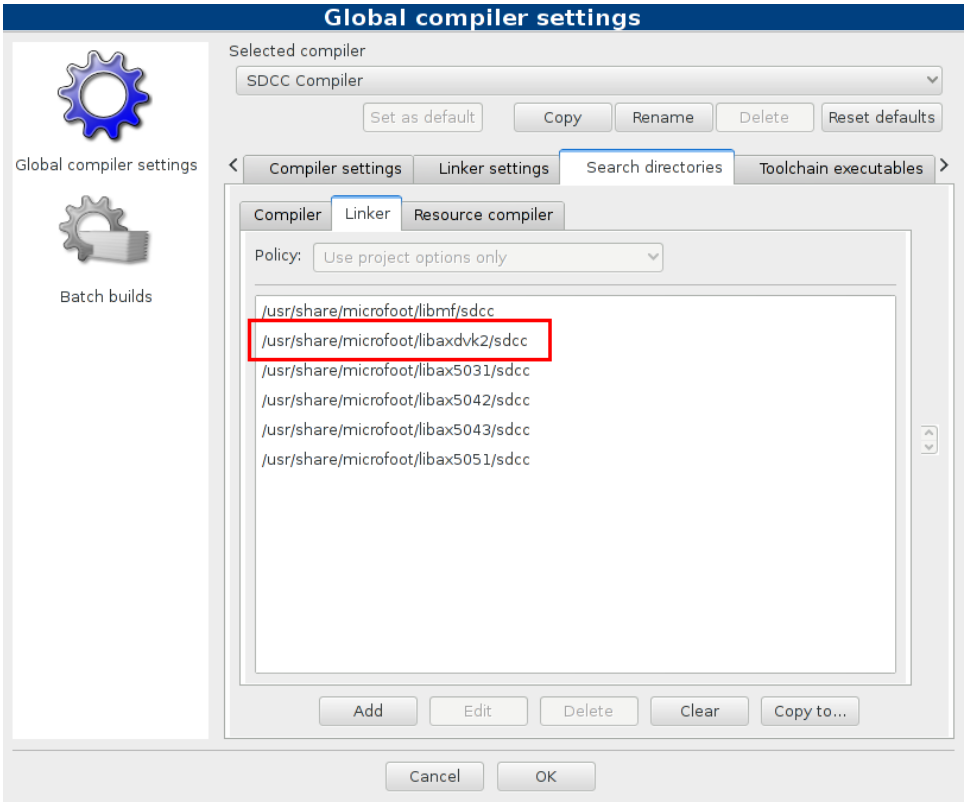
4. Evaluation Board Peripherals

In order to better separate the universally useful routines of LibMF from the routines only useful for the Axsem AX-DVK2 evaluation board, the latter routines will be removed from LibMF in the future. Therefore, the functions in this chapter are deprecated and should no longer be used.

Instead, a new library called libaxdvk2 will provide support routines for the AX-DVK2 board. The LCD routines are also provided by libaxdvk2. In order to convert existing projects to use the LCD routines of libaxdvk2, perform the following steps:

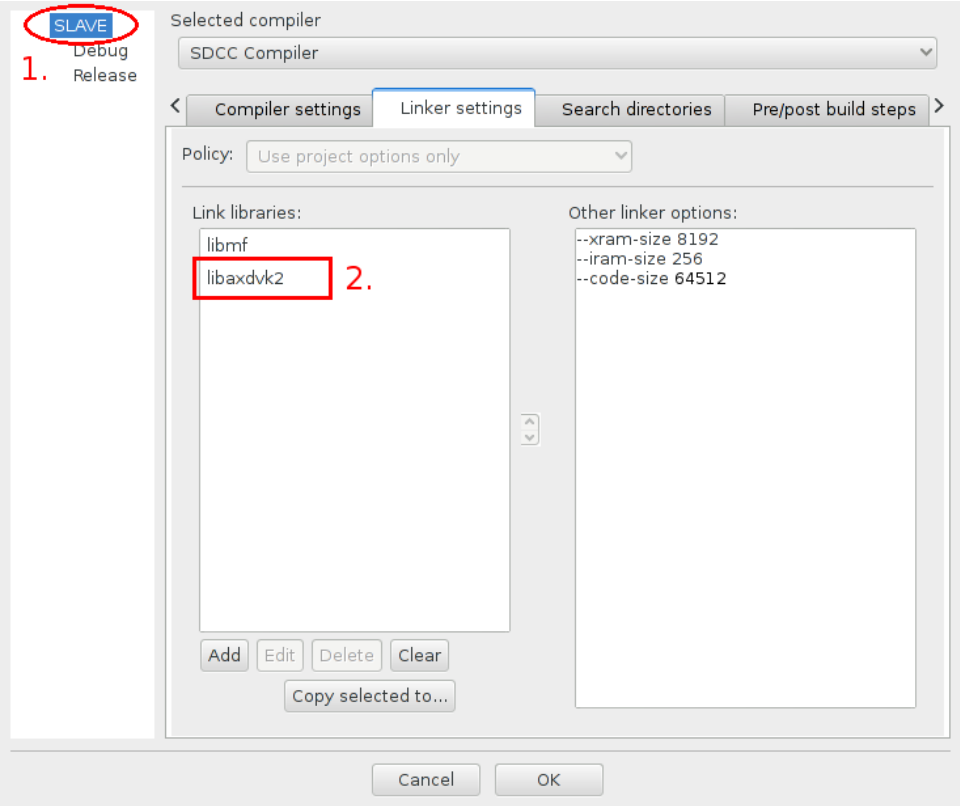
1. For existing AxCode::Blocks installations, add the include and library paths of libaxdvk2 to the global compiler settings. New installations will have the paths automatically configured. Open Settings → Compiler, click on the Search directories tab, and add the paths to the Compiler and Linker sub-tab as illustrated in the following two screen-shots:





2. For existing projects that use libmflcd.h, replace the include statements for libmflcd.h with include statements for libaxlcd.h

3. For existing projects for the AX-DVK2 evaluation board, add libaxdvk2 to the list of libraries to be linked. Open Project → Build Options, click on the Linker settings tab, select the root of the tree in the left pane of the window (marked by 1), and add libaxdvk2 to the list of libraries in the right pane (marked by 2).



4.1. LIBMFLCD.H

libmflcd.h contains routines for accessing the alphanumeric 2x16 character liquid crystal display (LCD) on the evaluation board. libmflcd.h routines are blocking and do not require an interrupt. The routines may take a very long time to complete due to the slowness of the LCD.

4.1.1. VOID LCD_INIT(VOID)

This function initializes the interface and resets the LC display and sets it up.

4.1.2. VOID LCD_PORTINIT(VOID)

This function only initializes the interface, but leaves the display alone. It can be used instead of lcd_init() when waking up from sleep or deepsleep while the display was kept powered, and it is undesirable if the display contents change.

4.1.3. void lcd_setpos(uint8_t v)

This function positions the LC display write cursor. Top line characters are numbered from left to right from 0x00 to 0x0F, and bottom line characters are numbered from 0x40 to 0x4F. The cursor does not auto-wrap from the top to the bottom line. After calling lcd_setpos(), lcd_waitshort() must be called or an equivalent delay must be kept before accessing the LCD again.

4.1.4. void lcd_writestr(const char *ch)

This function writes the null terminated C string pointed to by ch to the display at the current write cursor location. \n causes the output to continue at the beginning of the second line.

4.1.5. void lcd_writehex16(uint16_t val, uint8_t nrdig, uint8_t flags)
void lcd_writehex32(uint32_t val, uint8_t nrdig, uint8_t flags)

These functions write a hexadecimal number (val) to the display. The number of digits is given by nrdig. Leading characters are filled with zeros. flags is a bitwise or combination of the WRNUM constants documented in section 2.3.19

4.1.6. void lcd_writenum16(uint16_t val, uint8_t nrdig, uint8_t flags)
void lcd_writenum32(uint32_t val, uint8_t nrdig, uint8_t flags)

These functions write a decimal number (val) to the display. The number of digits is given by nrdig. flags is a bitwise or combination of the WRNUM constants documented in section 2.3.19

4.1.7. void lcd_cleardisplay(void)

This function clears the complete LC Display. It must be followed by lcd_waitlong, or the equivalent delay, before accessing the LC Display again.

4.1.8. void lcd_clear(uint8_t pos, uint8_t len)

This function clears len characters starting at position pos.

4.1.9. void lcd_waitlong(void)
void lcd_waitshort(void)

The LC display is very slow processing commands and data. These routines delay the microprocessor by a certain time by busy waiting. One of those routines, or the equivalent delay, should be called after each LC command or character. Which LC commands require the long delay is detailed in the LC display datasheet.

4.1.10. void lcd_writecmd(uint8_t cmd)

This low level routine writes a command to the display.

4.1.11. void lcd_writedata(uint8_t d)

This low level routine writes a character to the display.

5. Contact Information

AXSEM AG
Oskar-Bider-Strasse 1
CH-8600 Dübendorf
SWITZERLAND

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

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