Wireless M-Bus Documentation

Wireless M-Bus HAL

STACKFORCE

embedded.connectivity.solutions

March 20, 2015

Contents

Chapter 1

Introduction

Welcome to a short introduction to the Wireless M-Bus Hardware Abstraction Layer. The main target for the HAL is on connecting the two core libraries *Wireless M-Bus stack* and *Si446x RF driver* with the hardware underneath. The HAL serves for solving all the hardware dependencies the stack and the RF driver have some hardware dependencies do have.

<table>
<tr><td colspan="2">Serial Layer/Customer Application</td></tr>
<tr><td colspan="2">Wireless M-Bus stack
(library)</td></tr>
<tr><td rowspan="3">WMBUS-HAL
Generic MCU AES MEM TMR UART</td><td>WMBUS-HAL
RF</td></tr>
<tr><td>Si446x RF driver
(library)</td></tr>
<tr><td>Si446x RF driver HAL
GPIO SPI MCU</td></tr>
<tr><td colspan="2">...hardware...</td></tr>
</table>

Chapter 2

Porting guide for EFM and EZR starter kit

Basic explanation

The demo projects are prepared for use with the following starter kits:

- SLWSTK6220A, based on EZR32WG (basically an EFM32WG + Si4460)
- STK3200, based on EFM32ZG
- STK3600, based on EFM32LG
- STK3800, based on EFM32WG

Base configuration is for building a meter device in mode S2. It uses libraries for the transceiver and for the stack.

Switching device type and mode

- choose the correct Wireless M-Bus stack (*libstack*) and Si446x RF driver (*librf*) libraries according to your needs:
  - Cortex family dependent on the MCU to be used, e.g.: EFM32ZG -> Cortex-M0+, EFM32LG -> Cortex-M3, EFM32WG -> Cortex-M4
  - Wireless M-Bus device type (meter/collector)
  - Wireless M-Bus mode (e.g. S2, T2, ...)
  - libraries can be found in `/libs/rf` and `/libs/stack`
- choose pre-include header file according to the device type and mode you would like to use:
  1. Within IAR navigate to *Options -> C/C++ Compiler -> Preprocessor -> Preinclude File*
  2. pre-include header files can be found in `/src/configs/`

Switching to another target

Within IAR do the following

- choose the MCU device to be used:
  - *Options -> General Options -> Device*
- change the appropriate include path to match the MCU family of your choice:
  - Example for EFM32LG (Leopard Gecko): `$PROJ_DIR$\..\..\..\src\hal\mcu\EFM32LIB\Device\EZM32WG\Include`
- change the defined symbol according to your kit:
  - Options->C/C++ Compiler->Preprocessor->Defined symbols
  - for the starter kits, the following switches are available: `SLWSTK6220A`, `STK3200`, `STK3600`, `STK3800`

Regarding the board configuration (e.g. pin-out, selected UART interface, ...)

- pin configuration can be changed by either modifying one of the existing configuration files for the starter kits (located at `/src/configs/boards`) or by creating and adding a new one to the same location. in case you're creating a new configuration file:
  - place it at the same location
  - create a new board configuration switch to the `prj_config.h`-file' within the application configuration you're using. Example: For creating a new board configuration for usage with the stack and the serial layer on top, you need to modify the following file and add the recently created board configuration file: `/src/configs/prj_config/app/serial/prj_config.h`
    - in case you add your own switch for this board, remember to activate that switch by adding it to the IAR project: *Options -> C/C++ Compiler -> Preprocessor -> Defined symbols*
- adapt the HAL-files if necessary

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Module Documentation

5.1 Wireless M-Bus Hardware Abstraction Layer (WMBUS-HAL)

5.1.1 Detailed Description

This is the main module for the HAL. Typically this module handles the initialization of all the other HAL modules only.

The file `/src/target/`sf_hal.c needs to be touched only in case additional HAL modules have been added and need to be initialized during startup.

5.1.2 Macro Definition Documentation

5.1.2.1 #define HAL_STATUS_OK(A) (A == **E_HAL_STATUS_SUCCESS**)

Simple macro for checking the value returned on initialization of each HAL module.

5.1.3 Enumeration Type Documentation

5.1.3.1 enum **E_HAL_STATUS_t**

Enumeration of HAL operation status.

Enumerator

E_HAL_STATUS_INVALID Invalid return status.

E_HAL_STATUS_SUCCESS Initialization has been successful.

E_HAL_STATUS_MCU_ERROR An error occurred in the MCU module.

E_HAL_STATUS_MEM_ERROR An error occurred in the MEM module.

E_HAL_STATUS_RF_ERROR An error occurred in the RF module.

E_HAL_STATUS_AES_ERROR An error occurred in the AES module.

E_HAL_STATUS_UPDATE_ERROR An error occurred in the UPDATE module.

E_HAL_STATUS_TMR_ERROR An error occurred in the TMR module.

E_HAL_STATUS_UART_ERROR An error occurred in the UART module.

E_HAL_STATUS_UNKNOWN_ERROR An unknown error occurred during initialization.

5.1.4 Function Documentation

5.1.4.1 **E_HAL_STATUS_t** wmbus_hal_init (void)

Initializes all hardware components.

This function must be called the main application at the very beginning and in order to initialize all the hardware prior to initializing the stack. Usually this function simply calls the initialization routines of all the other more specific HAL modules and returns an error if one them fails.
Returns

Status of the operation.

5.2 WMBUS-HAL - Hardware accelerated AES

5.2.1 Detailed Description

This module defines the Hardware Abstraction Layer for the hardware accelerated Advanced Encryption Standard (AES).

All the EFM/EZR devices do provide an AES hardware acceleration. This HAL module abstracts access to the EFM/↩ EZR hardware for the Wireless M-Bus Stack. The implementations can be found in file `/src/target/`sf_hal_aes.c.

5.2.2 Macro Definition Documentation

5.2.2.1 #define EFM_AES_BLOCK_LEN (16U)

Specifies the length of encryption blocks to be processed by the AES module.
Warning

This value is fixed for AES and must not be changed!

5.2.2.2 #define EFM_AES_KEY_LEN (16U)

Specifies the length of the key to be used with AES module by EFM/EZR devices.
Warning

This value should not be changed, unless you know what you're doing!

5.2.3 Function Documentation

5.2.3.1 bool_t wmbus_hal_aes_cbcDecrypt (const uint8_t * pc_in, uint8_t * pc_out, uint16_t n_block, uint8_t * pc_iv)

Performs *CBC decryption* on a given number of blocks. This function performs *CBC decryption* on a given number of blocks, using the given initialization vector (iv).

Parameters

pc_in	Pointer to the data to be decrypted.

pc_out	Pointer where to store the decrypted data.
n_block	Number of blocks to be decrypted.
pc_iv	Buffer holding 128 bit initialization vector to be used.

Returns

Returns `TRUE` if data has been decrypted successfully. In case of any problem (e.g. missing key) it'll return `FALSE`.

5.2.3.2 bool_t wmbus_hal_aes_cbcEncrypt (const uint8_t * pc_in, uint8_t * pc_out, uint16_t n_block, uint8_t * pc_iv)

Performs *CBC encryption* on a given number of blocks. This function performs *CBC encryption* on a given number of blocks, using the given initialization vector (iv).

Parameters

pc_in	Pointer to the data to be encrypted.
pc_out	Pointer where to store the encrypted data.
n_block	Number of blocks to be encrypted.
pc_iv	Buffer holding 128 bit initialization vector to be used.

Returns

Returns `TRUE` if data has been encrypted successfully. In case of any problem (e.g. missing key) it'll return `FALSE`.

5.2.3.3 bool_t wmbus_hal_aes_ctrDecrypt (const uint8_t * pc_in, uint8_t * pc_out, uint8_t c_len, uint8_t * pc_iv)

Performs *CTR decryption* on a given number of blocks. This function performs *CTR decryption* on a given number of blocks, using the given initialization vector (iv).

Parameters

pc_in	Pointer to the data to be decrypted.
pc_out	Pointer where to store the decrypted data.
n_block	Number of blocks to be decrypted.
pc_iv	Buffer holding 128 bit initialization vector to be used.

Returns

Returns `TRUE` if data has been decrypted successfully. In case of any problem (e.g. missing key) it'll return `FALSE`.

5.2.3.4 bool_t wmbus_hal_aes_ctrEncrypt (const uint8_t * pc_in, uint8_t * pc_out, uint8_t c_len, uint8_t * pc_iv)

Performs *CTR encryption* on a given number of blocks. This function performs *CTR encryption* on a given number of blocks, using the given initialization vector (iv).

Parameters

pc_in	Pointer to the data to be encrypted.
pc_out	Pointer where to store the encrypted data.
n_block	Number of blocks to be encrypted.
pc_iv	Buffer holding 128 bit initialization vector to be used.

Returns

Returns `TRUE` if data has been encrypted successfully. In case of any problem (e.g. missing key) it'll return `FALSE`.

5.2.3.5 bool_t wmbus_hal_aes_decrypt (const uint8_t * pc_in, uint8_t * pc_out)

Decrypts a single block only. Decrypts a single block only, using the key that is currently stored within the HAL module. Block length for AES is fixed to 16 bytes.

Parameters

pc_in	Pointer containing data to be decrypted.
pc_out	Pointer where to store the decrypted data.

Returns

Returns `TRUE` if data has been decrypted successfully. In case of any problem (e.g. missing key) it'll return `FALSE`.

5.2.3.6 bool_t wmbus_hal_aes_encrypt (const uint8_t * pc_in, uint8_t * pc_out)

Encrypts a single block only. Encrypts a single block only, using the key that is currently stored within the HAL module. Block length for AES is fixed to 16 bytes.

Parameters

pc_in	Pointer to the data to be encrypted.
pc_out	Pointer where to store the encrypted data.

Returns

Returns `TRUE` if data has been encrypted successfully. In case of any problem (e.g. missing key) it'll return `FALSE`.

5.2.3.7 bool_t wmbus_hal_aes_init (void)

Performs the initialization of the AES HAL module. This function usually will be called by wmbus_hal_init() and enables the AES HAL module to initialize the hardware.

Returns

On successful initialization the function should return `TRUE`. Just in case the initialization wasn't possible for any reason, this function should return `FALSE`.

5.2.3.8 bool_t wmbus_hal_aes_setKey (const uint8_t * pc_key)

Sets key for en-/decryption. Sets key to be used for any encryption or decryption procedure by the AES HAL module. Basically the key will be maintained by the stack and given to the AES HAL module for maintaing the underlying hardware. There is no need to call this function by user application.

Parameters

pc_key	Pointer to the buffer which is holding the key.

Returns

TRUE, if key was set successfully, FALSE otherwise

5.3 WMBUS-HAL - MCU core

5.3.1 Detailed Description

The HAL module for the Microcontroller Unit (MCU) is for handling functions that are closely related to the MCU core.

All these functions are implemented in `/src/target/`sf_hal_mcu.c

5.3.2 Function Documentation

5.3.2.1 bool_t wmbus_hal_mcu_init (void)

Initializes the MCU core. In case the MCU core needs some kind of initialization (e.g. setting up the core clock, ...), this will be handled within the MCU initialization.

Includes required by the EFM librarry Verification if macro specifying the MCU speed has been set.
Returns

Returns `FALSE` in case initialization failed.

First, globally disabling interrupts

Then applying, chip errata

Set the clock speed

Enabled GPIO clock

Finally, globally re-enabling interrupts

5.3.2.2 void wmbus_hal_mcu_reset (void)

Resets the MCU. This function handles the proper reset/restart of the complete MCU.

Forcing a system reset by using the system reset request provided by the EFM library.

5.4 WMBUS-HAL - Non-volatile memory

5.4.1 Detailed Description

This module defines the Hardware Abstraction Layer for the non-volatile memory unit. The stack requires non-volatile for storing data that should be kept even if the device looses power supply, e.g. the currently configured address of the device.

All these functions are implemented in `/src/target/`sf_hal_mem.c

5.4.2 Macro Definition Documentation

5.4.2.1 #define MEM_END_ADDR (**MEM_START_ADDR** + FLASH_PAGE_SIZE)

Setting macro for end address of flash.

5.4.2.2 #define MEM_FLASH_WRITE_RETRIES (3U)

Check if FLASH_PAGE_SIZE macro has been set by EFM library. Check if FLASH_SIZE macro has been set by EFM library.

In case flash access fails, the driver retries it. The number of retries can be specified by setting the MEM_FLASH_↩ WRITE_RETRIES macro.

5.4.2.3 #define MEM_START_ADDR (FLASH_SIZE - FLASH_PAGE_SIZE)

Setting macro for start address of flash to be used as non-volatile memory for the stack. Usually using the very last flash page is preferred.

5.4.3 Function Documentation

5.4.3.1 uint16_t loc_datamemory_writeData (uint32_t l_addr, uint8_t * pc_data, uint16_t i_len)

Handles writing of data. This function handles the write procedure to the flash.

Parameters

l_addr	Address within flash where to write data.
pc_data	Pointer to the data to be written into flash.
i_len	Number of bytes to copied from pc_data to l_addr.

Returns

Returns the number of written bytes.

Return values

(0)	Returns zero in case write operation has failed.

5.4.3.2 bool_t wmbus_hal_mem_init (void)

Initializes the memory module, in case the non-volatile needs any preparation before using it.

Returns

Returns `TRUE` if initialization has been successful.

5.4.3.3 uint16_t wmbus_hal_mem_read (uint8_t * pc_data, uint16_t i_len, uint32_t l_offset)

Reads data from the non-volatile memory. Especially on startup of the device, the stack needs to read importand data that has been saved earlier (e.g. device address).

Parameters

pc_data	Destination where to copy the read bytes.
i_len	Number of bytes to be read.
l_offset	Offset from start address of the non-volatile memory, equivalent to a virtual mapped address.

Returns

Number of bytes that have been read.

5.4.3.4 uint16_t wmbus_hal_mem_write (uint8_t * pc_data, uint16_t i_len, uint32_t l_offset)

Writes data into the non-volatile memory. This function provides access to the non-volatile memory for storing stack internal information that need to be kept in case of power loss.

Parameters

pc_data	Data to be written into the memory.
i_len	Number of bytes to be written.
l_offset	Offset from start address of the non-volatile memory, equivalent to a virtual mapped address.

Returns

Number of bytes that have been written.

5.5 WMBUS-HAL - RF driver interface

5.5.1 Detailed Description

This module defines the Hardware Abstraction Layer for a radio driver. Please note, that the radio driver itself is responsible to define hardware requirements as communication interfaces respectively pins and ports of the MCU for control.

This is the implementation of the RF driver interface. However, as this delivery is designed for Silabs products only, all the RF driver interfaces are simply forwarded to the Si446x RF driver library. The implementation can be found in `/src/target/`sf_hal_rf.c.

5.5.2 Macro Definition Documentation

5.5.2.1 #define HAL_RF_MODE_C_FRAMETYPE_A 0xCDU

Mode C preamble for frame type A

5.5.2.2 #define HAL_RF_MODE_C_FRAMETYPE_B 0x3DU

Mode C preamble for frame type B

5.5.2.3 #define HAL_RF_MODE_C_PREAMBLE_FIRST 0x54U

First byte of mode C preamble

5.5.2.4 #define HAL_RF_MODE_N_SYNC1 0xF6U

First byte of mode N sync word

5.5.2.5 #define HAL_RF_MODE_N_SYNC2_FRAME_A 0x8DU

Mode C preamble Frame type A. Second sync byte for A.

5.5.2.6 #define HAL_RF_MODE_N_SYNC2_FRAME_B 0x72U

Mode C preamble Frame type B. Second sync byte for B.

5.5.2.7 #define HAL_RF_NEW_TLG 0xFFFFU

This value indicates the beginning of the reception of a brand new telegram.

5.5.2.8 #define HAL_RF_QUALITY_LEN 2U

These are macros required for usage within RX callback function fp_hal_rf_evt_rx which reports receptions to the PHY layer. They must not be changed. Number of link quality bytes. Used within wmbus_hal_rf_rxInit.

5.5.3 Typedef Documentation

5.5.3.1 typedef void(* fp_hal_rf_evt_rx)(uint16_t i_len, E_WMBUS_FRAME_t e_frameType)

Callback which indicates a new reception.

Parameters

i_len	Length of the received data.
e_frameType	Specifying the type of Wireless M-Bus frame.

5.5.3.2 typedef void(* fp_hal_rf_evt_tx)(uint16_t i_len)

Callback which indicates a finished transmission.

Parameters

i_len	Length of the transmitted data.

5.5.4 Enumeration Type Documentation

5.5.4.1 enum **E_HAL_RF_CALIBRATE_t**

Enumeration required for wmbus_hal_rf_reset function specifying if the RF driver shall calibrate the transceiver after reset or not.

Enumerator

E_HAL_RF_CALIB_CALIBRATE_STORE Calibrate and store register values.

E_HAL_RF_CALIB_LOAD Load stored values to registers.

E_HAL_RF_CALIB_OFF No calibration as powering down.

5.5.4.2 enum **E_HAL_RF_CS_STATUS_t**

Enumeration of the status of a carrier sense.

Enumerator

E_HAL_RF_CS_STATUS_NO_CARRIER_DETECTED The transmission medium is free, no carrier detected.

E_HAL_RF_CS_STATUS_CARRIER_DETECTED The transmission medium is occupied, carrier detected.

E_HAL_RF_CS_STATUS_INVALID_STATE The chip is in an invalid state to perform carrier sense.

E_HAL_RF_CS_STATUS_ERROR An error occured. CS could not be performed.

5.5.4.3 enum **E_HAL_RF_MODE_t**

Enumeration for the wmbus_hal_rf_rxFinish function, specifying on how to proceed after finishing RX procedure.

Enumerator

E_HAL_RF_MODE_RUN Running mode

E_HAL_RF_MODE_WAIT Waiting

5.5.4.4 enum **E_HAL_RF_POSTAMBLE_t**

Possible value for postamble that may be used by the RF driver after transmission.

Enumerator

E_HAL_RF_POSTAMBLE_NONE No postamble to send.

E_HAL_RF_POSTAMBLE_ODD 0101 postamble to send.

E_HAL_RF_POSTAMBLE_EVEN 1010 postamble to send.

5.5.4.5 enum **E_HAL_RF_POWERMODE_t**

Enumeration of low power modi for the RF driver that are maintained by the stack.

Enumerator

E_HAL_RF_POWERMODE_OFF Shutdown mode

E_HAL_RF_POWERMODE_IDLE Low power, register settings kept

E_HAL_RF_POWERMODE_RX Active power mode for Rx and Tx

E_HAL_RF_POWERMODE_MAX Active power mode

5.5.5 Function Documentation

5.5.5.1 **E_HAL_RF_CS_STATUS_t** wmbus_hal_rf_carrierSense (sint8_t c_rssiThres)

Listens to the medium and reports whether its free or not. The functions uses the chip internal features to perform carrier sense. Before calling this function, the chip itself and the SPI communication interface must be initialized. The function sets the chip in the correct state to perform carrier sense.

Parameters

sint8_t	The RSSI threshold in two's complement. After the RSSI is valid, the value is compared with this threshold. When the RSSI is bigger as the threshold, the chip detects a carrier.

Return values

E_HAL_RF_CS_STATUS_NO_CARRIER_DETECTED	No carrier detected.
E_HAL_RF_CS_STATUS_CARRIER_DETECTED	Carrier detected.
E_HAL_RF_CS_STATUS_INVALID_STATE	The chip is in an invalid state.
E_HAL_RF_CS_STATUS_ERROR	An error occured and CS could not be performed.

5.5.5.2 uint16_t wmbus_hal_rf_getRfChannel (void)

Returns the currently channel of the RF communication.
Returns

Current channel.

5.5.5.3 E_WMBUS_MODE_t wmbus_hal_rf_getRxSenseTuning (void)

Returns the current transceiver RX configuration. Returns the current transceiver RX configuration whether it is tuned for reception of either mode T or mode C meters. Please note that this feature is optional as it is limited to be used for mode C collectors only. Additionally, this feature must be supported by the underlying RF driver. You may ask your support contact for further information on this feature. The tuning configuration can be set by using wmbus_hal_rf_setRxSenseTuning.

Returns

This can be either E_WMBUS_MODE_T or E_WMBUS_MODE_C.

5.5.5.4 uint8_t wmbus_hal_rf_getSignalStrength (void)

Get the signal strength of the transceiver. Requests the RF driver to read the currently configured output power.
Returns

Current signal strength from -130dBm (0x0) to 125dBm (0xFE).

Return values

0xFF	The value is invalid, maybe because this isn't supported by the transceiver or an invalid value has been read.

5.5.5.5 uint16_t wmbus_hal_rf_getTelegramDelay (void)

Retrieves the delay of receiving a telegram. Will be called for retrieving the delay in 500us steps the transceiver needs for receiving and forcing the data into the stack.
Note

This function is required for some RF drivers only.
Returns

Timeout in 500us steps Example: 1 = 500us, 3 = 1,5ms

5.5.5.6 bool_t wmbus_hal_rf_init (void)

Initializes the RF module. This function simply initializes the RF driver itself. Before any communication starts, the stack will call wmbus_hal_rf_start.
Returns

Returns `TRUE` if initialisation was successful.

5.5.5.7 bool_t wmbus_hal_rf_reset (**E_HAL_RF_CALIBRATE_t** e_calibrate)

Resets the RF module. The stack will request the RF driver to reset any time the stack assumes the RF driver needs to re-enter a defined state.

Parameters

e_calibrate	Specifies if the RF driver shall calibrate (and store values), simply load existing values or omit calibration.

Returns

Returns `TRUE` when reset has been performed successfully.

5.5.5.8 bool_t wmbus_hal_rf_rxData (uint8_t * pc_data, uint16_t i_len)

Retrieves a chunk of data from the RF driver. Just like the transmission, reception of data is done chunk-wise. For this, the function will be called as fast as possible to retrieve all data received by the transceiver.

Parameters

pc_data	Buffer to write the received data into.
i_len	Number of bytes to receive.

Returns

Returns `TRUE` if the data could be received successfully.

5.5.5.9 bool_t wmbus_hal_rf_rxFinish (**E_HAL_RF_MODE_t** e_mode)

Finalises the reception of data. At least wmbus_rf_rxInit should be called before.

Parameters

e_mode	Indicates the status the transceiver shall enter: E_HAL_RF_MODE_WAIT: cleans the buffers and set the transceiver to IDLE state. E_HAL_RF_MODE_RUN: re-enter RX state for further receptions.

Returns

Returns `TRUE` if the finalisation was successful.

5.5.5.10 bool_t wmbus_hal_rf_rxInit (uint8_t * pc_quality, uint8_t c_len)

Initiates the reception of data. As soon as the stack has been informed about the detection of a telegram, the stack will initialise the reception procedure by calling this function.

Parameters

pc_quality	Pointer to the memory where to store the link quality. The number of quality values may differ between the RF modules. The maximum number of bytes for the quality field is specified by `RF_QUALITY_LEN`. The parameter will be NULL if link quality is not provided by the RF driver. Otherwise, the first byte is always the RSSI value: 0xFF no link quality available FE -254 dBm ... 00 0 dBm
c_len	Number of quality bytes to read.

Returns

Returns `FALSE` if the receiving parameters could not be set successfully.

5.5.5.11 bool_t wmbus_hal_rf_setCallback (**fp_hal_rf_evt_tx** fp_tx, **fp_hal_rf_evt_rx** fp_rx)

Sets the callback pointer for TX and RX events.

Parameters

fp_tx	Function pointer to the tx handle function.
fp_rx	Function pointer to the rx handle function.

Returns

Returns `TRUE` if initialisation has been succesful.

5.5.5.12 bool_t wmbus_hal_rf_setDataRate (E_WMBUS_DATA_RATE_t e_dataRate)

Sets the data rate of the transceiver. Requests the RF driver to configure the transceiver for the requested data rate on transmission and receptions.

Parameters

e_dataRate	Data rate to set.

Returns

bool_t Returns `TRUE` if the data rate was set successfully.

5.5.5.13 bool_t wmbus_hal_rf_setFrequencyOffset (sint16_t si_freqOffset)

Sets the frequency offset of the carrier.

Parameters

si_freqOffset	Frequency offset in kHz.

Returns

Returns `TRUE` if successful.

5.5.5.14 bool_t wmbus_hal_rf_setPowerMode (**E_HAL_RF_POWERMODE_t** e_powermode)

Sets the power mode of the RF driver.

Parameters

e_powermode	An enumerated value for the wanted mode.

Returns

Returns `TRUE` if successful.

5.5.5.15 bool_t wmbus_hal_rf_setRfChannel (uint16_t i_channel)

Sets the channel to use for RF communication.

Parameters

i_channel	Channel to set.

Returns

Returns `TRUE` if successful.

5.5.5.16 bool_t wmbus_hal_rf_setRxSenseTuning (E_WMBUS_MODE_t e_mode)

Enables mode C collector RX configuration for receiving mode T and C. Please note that this feature is optional as it is limited to be used for mode C collectors only. Additionally, this feature must be supported by the underlying RF driver. You may ask your support contact for further information on this feature. The current tuning configuration can be requested by using wmbus_hal_rf_getRxSenseTuning.

Parameters

e_mode	This can be either E_WMBUS_MODE_T or E_WMBUS_MODE_C to tune for for mode T or mode C meters.

Returns

Returns TRUE if tuning was successful.

5.5.5.17 bool_t wmbus_hal_rf_setSignalStrength (uint8_t c_signal)

Sets the signal strength for transmission. Requests the RF driver to set the output power for the transceiver to the appropriate value.

Parameters

c_signal	Signal strength from -130dBm (0x0) to 125dBm (0xFE). 0xFF is reserved. If c_signal exceeds the minimum or maximum TX power settings the output power will be set to the lowest respective highest supported value.

5.5.5.18 bool_t wmbus_hal_rf_txData (uint8_t * pc_data, uint16_t i_len)

Transmits a chunk of data. For Wireless M-Bus the data will be sent chunk wise. Therefore all data to be transmitted will be delivered to the RF driver by sequentially calling this function.
Warning

Prior to calling this function, wmbus_hal_rf_txInit must be called

Parameters

pc_data	Pointer to data to be transmitted.
i_len	Number of bytes to be transmitted.

Returns

Returns whether transmission was successful or failed.

5.5.5.19 bool_t wmbus_hal_rf_txFinish (void)

Finalises the transmission of data. This function will be called by the stack to finalise a transmission, independently if transmission succeeded of failed. At least wmbus_rf_txInit should be called before first.

Returns

Returns whether finalisation was successful or failed.

5.5.5.20 bool_t wmbus_hal_rf_txInit (uint16_t i_len, E_WMBUS_FRAME_t e_frameType, E_WMBUS_MODE_t e_mode)

Initiates the transmission of data. This function will be called for initialising the transmission procedure.

Parameters

i_len	Total number of bytes to be sent.
e_frameType	Specifies the Wireless M-Bus frame type to be used for this transmission.
e_mode	Specifies the Wireless M-Bus mode to be used. In S and T-Mode this parameter will not be observed from the lower layers. In C-mode this parameter informs the lower layers which RF configuration shall be used to communicate.

Returns

Returns whether TX initialisation was successful or failed.

5.5.5.21 void wmbus_hal_rf_txSetPostamble (**E_HAL_RF_POSTAMBLE_t** e_postamble)

Sets the postamble to be used for transmissions.

Parameters

e_postamble	Specifies the specific value of postamble to be used.

5.6 WMBUS-HAL - Hardware timer interface

5.6.1 Detailed Description

This HAL module shall provide a hardware timer for the stack internally maintained timer module and is mandatory and very important for proper functioning of the stack.

All these functions are implemented in `/src/target/`sf_hal_tmr.c

5.6.2 Macro Definition Documentation

5.6.2.1 #define TMR_DEFAULT

Value:

```
{   true,                   /* Enable timer when init complete. */                    \
    false,                  /* Stop counter during debug halt. */                     \
    timerPrescale1,         /* No prescaling. */                                      \
    timerClkSelHFPerClk,    /* Select HFPER clock. */                                 \
    timerInputActionNone,   /* No action on falling input edge. */                    \
    timerInputActionNone,   /* No action on rising input edge. */                     \
    timerModeUp,            /* Up-counting. */                                        \
    false,                  /* Do not clear DMA requests when DMA channel is active. */ \
    false,                  /* Select X2 quadrature decode mode (if used). */         \
    false,                  /* Continuous mode. false:diable one shot mode */         \
    false                   /* Not started/stopped/reloaded by other timers. */       \
}
```

Timer settings required for the stack.
Warning

> As the timer is very important to the stack, do not changes these settings unless you know what you're doing.

5.6.3 Function Documentation

5.6.3.1 bool_t wmbus_hal_tmr_init (uint16_t i_ticksPerSecond)

Initializes the timer. Requests the driver for the hardware timer to initialize for timer interrupt at an interval of the given value.

Parameters

i_ticksPerSecond	Specifies the number of timer tick indications to be done per second.

Returns

Returns TRUE if initialisation succeeded.

5.6.3.2 bool_t wmbus_hal_tmr_offset (sint16_t si_offset)

Adds a positive or negative offset to the timer tick counter.

Parameters

si_offset	Offset to be applied on the current tick counter.

Returns

Returns TRUE if successful.

5.6.3.3 bool_t wmbus_hal_tmr_set (uint16_t ui_counterValue)

Sets the timer tick counter. For the dynamic timer function, the stack may request the hardware timer driver to set the tick counter to the requested value.

Parameters

ui_counterValue	Tick counter value to be set.

Returns

Returns TRUE if successful.

5.6.3.4 bool_t wmbus_hal_tmr_setCallback (**fp_hal_tmr_cb** fp_tmr)

Sets the callback for the timer event. Provides a pointer to the function to be called, when the hardware timer is causing an interrupt due to the configured interval.

Parameters

fp_tmr	Pointer to the callback function.

Returns

Returns TRUE if setting the callback has been successful.

5.7 WMBUS-HAL - Serial interface

5.7.1 Detailed Description

This module defines the Hardware Abstraction Layer for the serial interface enabling M2M communication, as required if the stack shall be accessed *remotely* via the serial layer.

This module is commonly implemented by a driver for the Universal Asynchronous Receiver Transmitter (UART) provided by almost every MCU.

For using an EFM32ZG device, a LEUART driver for serial communication with the stack has been prepared: `/src/target/`sf_hal_leuart.c

Many of the EFM32 devices do have common UART-USB interface. There is a driver prepared for using UART-USB for serial communication with the stack included: `/src/target/`sf_hal_uartUsb.c

Recently added against the background of EZR devices is a driver for using the USART-USB interface: `/src/target/`sf_hal_usartUsb.c

5.7.2 Macro Definition Documentation

5.7.2.1 #define UART_BUFFER_RX_LEN 300U

Sets the length of the Rx ringbuffer.

5.7.2.2 #define UART_BUFFER_RX_LEN 300U

Sets the length of the Rx ringbuffer.

5.7.2.3 #define UART_BUFFER_RX_LEN 20U

EFM32LIB includes

Sets the length of the Rx ringbuffer.

5.7.2.4 #define UART_BUFFER_TX_LEN 50U

Sets the length of the Tx ringbuffer.

5.7.2.5 #define UART_USE_TX_IRQ TRUE

Enable UART Tx interrupts.

5.7.2.6 #define USB_BUFFER_TX_LEN 300U

Sets the length of the Tx ringbuffer.

5.7.2.7 #define USB_BUFFER_TX_LEN 300U

Sets the length of the Tx ringbuffer.

5.7.2.8 #define USB_RX_BUF_SIZ USB_FS_BULK_EP_MAXSIZE

EFM32LIB includes

Setup UART interface

5.7.2.9 #define USB_RX_BUF_SIZ USB_FS_BULK_EP_MAXSIZE

EFM32LIB includes

Setup UART interface

5.7.3 Enumeration Type Documentation

5.7.3.1 enum **E_HAL_UART_DATABITS_t**

Enumeration of all selectable data bits.

Enumerator

E_HAL_UART_DATABITS_8 8 databits

E_HAL_UART_DATABITS_7 7 databits

5.7.3.2 enum **E_HAL_UART_PARITY_t**

Enumeration of all selectable parity bits.

Enumerator

E_HAL_UART_PARITY_NONE No parity

E_HAL_UART_PARITY_EVEN Even parity

E_HAL_UART_PARITY_ODD Odd parity

Enumerator

E_HAL_UART_STOPBITS_0

5.7.3.3 enum **E_HAL_UART_STOPBITS_t**

Enumeration of all selectable stop bits.

	0 stopbits
E_HAL_UART_STOPBITS_1	1 stopbits
E_HAL_UART_STOPBITS_15	1,5 stopbits
E_HAL_UART_STOPBITS_2	2 stopbits

5.7.4 Function Documentation

5.7.4.1 EFM32_PACK_START (1)

Baudrate

Stop bits, 0=1 1=1.5 2=2

0=None 1=Odd 2=Even 3=Mark 4=Space

5, 6, 7, 8 or 16

To ensure size is a multiple of 4 bytes

Baudrate

Stop bits, 0=1 1=1.5 2=2

0=None 1=Odd 2=Even 3=Mark 4=Space

5, 6, 7, 8 or 16

To ensure size is a multiple of 4 bytes

5.7.4.2 void loc_sf_uart_rx_isr (void)

If the Rx-ringbuffer is full, disable the Rx-interrupt.

Otherwise read the next byte into the Rx-ringbuffer.

Increase the number of bytes in Rx-ringbuffer.

Check for an overflow of the read pointer and adjust if required.

Reset the Rx-interrupt.

5.7.4.3 void loc_sf_uart_tx_isr (void)

Reset the Tx-interrupt.

Decrease the number of bytes in Tx-ringbuffer.

If the Tx-ringbuffer is empty, disable the Tx-interrupt.

Otherwise write the next byte from Tx-ringbuffer.

Check for an overflow of the write pointer and adjust if required.

5.7.4.4 uint16_t wmbus_hal_uart_cntRxBytes (void)

Returns the number of ready-to-be-fetched bytes. This function is called by the stack in preparation to a receive operation and is for reading the number of received bytes, that are buffered by the stack.
Returns

Number of received bytes.

5.7.4.5 uint16_t wmbus_hal_uart_cntTxBytes (void)

Returns the current capability of transmitting bytes. In preparation to a transmit operation, the stack may request the number of bytes that could be transmitted. The UART driver shall return the number of bytes that possibly could be buffered until real transmission.
Returns

Number of bytes that could possibly be transmitted.

5.7.4.6 void wmbus_hal_uart_com_TxFinish (void)

Signals that a serial telegram is complete. This function will be called by the serial layer, as soon as a complete serial telegram has been sent using the wmbus_hal_uart_write function.

Decrease the number of bytes in ringbuffer.

Check for an overflow of the write pointer and adjust if required.

Decrease the number of bytes in ringbuffer.

Check for an overflow of the write pointer and adjust if required.

5.7.4.7 bool_t wmbus_hal_uart_init (void)

Initializes the UART. Within this function commonly the underlying UART interface will be configured accordingly.

Note

In case of projects around the Wireless M-Bus stack by STACKFORCE, usually the settings are: 115200 baud, 8 databits, no parity, 1 stopbit.

Returns

Returns `TRUE` if initialization has been successful.

Init the Rx-buffer variables.

Init the Tx-buffer variables.

Init the Tx-buffer variables.

Init the Tx-buffer variables.

5.7.4.8 bool_t wmbus_hal_uart_isRxOverflow (void)

Checks if there was a RX buffer overflow. This function will be called by the stack to double check if bytes have been lost, e.g. in case the stack has been receiving not plausible data. The UART driver should use this function to also reset its RX overflow error.

Returns

Returns `TRUE` if there was an overflow.

5.7.4.9 uint16_t wmbus_hal_uart_read (uint8_t * pc_data, uint16_t i_len)

Reads data from the UART. This function retrieves the requested amount of data from the UART driver.

Parameters

pc_data	Pointer to the location where to store received data.
i_len	Number of bytes to be read.

Returns

Number of successfully read bytes.

Read from Rx-ringbuffer until i_len or UART_BUFFER_RX_LEN is reached.

Write to the specified data pointer and increase the read pointer.

Decrease the number of bytes in ringbuffer.

Check for an overflow of the read pointer and adjust if required.

(Re-)Enable the Rx-interrupt.

Read from Rx-ringbuffer until i_len or UART_BUFFER_RX_LEN is reached.

Write to the specified data pointer and increase the read pointer.

Check for an overflow of the read pointer and adjust if required.

Read from Rx-ringbuffer until i_len or UART_BUFFER_RX_LEN is reached.

Write to the specified data pointer and increase the read pointer.

Check for an overflow of the read pointer and adjust if required.

5.7.4.10 uint16_t wmbus_hal_uart_write (uint8_t * pc_data, uint16_t i_len)

Writes data to the UART. This function will be called by the serial layer of the stack when stack requests to send data to the connected host.

Parameters

pc_data	Pointer to the data to be sent.
i_len	Number of bytes to be sent.

Returns

Returns the number of bytes successfully sent.

Write to Tx-ringbuffer until i_len or USB_BUFFER_TX_LEN is reached.

Write current byte to the write pointers address within the ringbuffer and increase the pointer.

Increase the number of bytes in ringbuffer.

Check for an overflow of the write pointer and adjust if required.

Write to Tx-ringbuffer until i_len or USB_BUFFER_TX_LEN is reached.

Write current byte to the write pointers address within the ringbuffer and increase the pointer.

Increase the number of bytes in ringbuffer.

Check for an overflow of the write pointer and adjust if required.

5.7.5 Variable Documentation

5.7.5.1 volatile bool_t gb_uart_bufferRxOverflow

Set if there was a Rx buffer overflow.

5.7.5.2 volatile uint8_t gc_uart_bufferRx[**UART_BUFFER_RX_LEN**]

Input ring buffer.

5.7.5.3 volatile uint8_t gc_uart_bufferTx[**UART_BUFFER_TX_LEN**]

Input ring buffer.

5.7.5.4 volatile uint16_t gi_uart_bufferRxLen

Number of bytes in the input buffer.

5.7.5.5 volatile uint16_t gi_uart_bufferTxLen

Number of bytes in the input buffer.

5.7.5.6 volatile uint8_t* gpc_uart_bufferRxRead

Pointer to the Rx ring buffer's current read position

5.7.5.7 volatile uint8_t* gpc_uart_bufferRxWrite

Pointer to the Rx ring buffer's current write position

5.7.5.8 volatile uint8_t* gpc_uart_bufferTxRead

Pointer to the Tx ring buffer's current read position

5.7.5.9 volatile uint8_t* gpc_uart_bufferTxWrite

Pointer to the Tx ring buffer's current write position

5.7.5.10 LEUART_Init_TypeDef usart_settings

Initial value:

```
=
{   leuartEnable,
    0,
    9600,
    leuartDatabits8,
    leuartNoParity,
    leuartStopbits1
}
```

5.8 Si446x driver GPIO HAL

5.8.1 Detailed Description

The GPIO HAL module provides access to the GPIO pins by the transceiver by abstracting the appropriate board and pin-out dependent MCU pins.

All the Si446x transceivers provide four GPIO pins (0-3). The RF driver uses only two of them. The two GPIO pins used are differentiated by GPIO0 and GPIO1. However, this HAL module may map the pin interactions to any of the four transceiver GPIO pins and may use the remaining two GPIO pins for other purposes (e.g. an external PA).

All the GPIO hardware abstraction for the RF driver is implemented within `/src/target/`sf_hal_gpio.c.

5.8.2 Typedef Documentation

5.8.2.1 typedef void(* fp_hal_gpio_radioIsr)(void)

Template for callback pointer for registering interrupt callback functions used for indicating an interrupt by one of the radio GPIO pins by the transceiver.

5.8.3 Function Documentation

5.8.3.1 void sf_hal_gpio_initGPIOx (void)

Initializes the MCU pins for accessing GPIO0 and GPIO1.

5.8.3.2 void sf_hal_gpio_irqClearFlagGPIO0 (void)

Clears the interrupt flag for GPIO0 pin.

5.8.3.3 void sf_hal_gpio_irqClearFlagGPIO1 (void)

Clears the interrupt flag for GPIO1 pin.

5.8.3.4 void sf_hal_gpio_irqDisableGPIO0 (void)

Disables the interrupt for GPIO0 pin.

5.8.3.5 void sf_hal_gpio_irqDisableGPIO1 (void)

Disables the interrupt for GPIO1 pin.

5.8.3.6 void sf_hal_gpio_irqEnableGPIO0 (void)

Enables the interrupt for GPIO0 pin.

5.8.3.7 void sf_hal_gpio_irqEnableGPIO1 (void)

Enables the interrupt for GPIO1 pin.

5.8.3.8 void sf_hal_gpio_irqFallingEdgeGPIO0 (void)

Sets the interrupt for the GPIO0 pin to *falling* edge sensitive.

5.8.3.9 void sf_hal_gpio_irqFallingEdgeGPIO1 (void)

Sets the interrupt for the GPIO1 pin to *falling* edge sensitive.

5.8.3.10 bool_t sf_hal_gpio_irqFlagIsSetGPIO0 (void)

Returns *TRUE* if the GPIO0 IRQ flag is set.

5.8.3.11 bool_t sf_hal_gpio_irqFlagIsSetGPIO1 (void)

Returns *TRUE* if the GPIO1 IRQ flag is set.

5.8.3.12 void sf_hal_gpio_irqRisingEdgeGPIO0 (void)

Sets the interrupt for the GPIO0 pin to *rising* edge sensitive.

5.8.3.13 void sf_hal_gpio_irqRisingEdgeGPIO1 (void)

Sets the interrupt for the GPIO1 pin to *rising* edge sensitive.

5.8.3.14 bool_t sf_hal_gpio_isHighGPIO0 (void)

Returns TRUE if GPIO0 pin level is *high*.

5.8.3.15 bool_t sf_hal_gpio_isHighGPIO1 (void)

Returns *TRUE* if GPIO1 pin level is high.

5.8.3.16 bool_t sf_hal_gpio_isLowGPIO0 (void)

Returns TRUE if GPIO0 pin level is *low*.

5.8.3.17 bool_t sf_hal_gpio_isLowGPIO1 (void)

Returns *TRUE* if GPIO1 pin level is low.

5.8.3.18 void sf_hal_gpio_modeRxSetGPIO0 (void)

This function is for setting the parameters and registers to enable the requested GPIO0 pin functionality in Rx Mode. In Rx Mode the GPIO0 pin signalizes that the Rx FIFOs almost full threshold has been reached by a *rising* edge.

5.8.3.19 void sf_hal_gpio_modeRxSetGPIO1 (void)

This function is for setting the parameters and registers to enable the requested GPIO1 pin functionality in Rx Mode. In Rx Mode the GPIO1 pin signalizes that the Rx FIFOs almost full threshold has been reached by a *rising* edge.

5.8.3.20 void sf_hal_gpio_modeTxSetGPIO0 (void)

This function is for setting the parameters and registers to enable the required GPIO0 PIN functionality in Tx Mode. In Tx Mode the GPIO0 pin signalizes that the Tx FIFOs almost full threshold has been reached by a *falling* edge.

5.8.3.21 void sf_hal_gpio_modeTxSetGPIO1 (void)

This function is for setting the parameters and registers to enable the required GPIO1 PIN functionality in Tx Mode. In Tx Mode the GPIO1 pin signalizes that the Tx FIFOs almost full threshold has been reached by a *falling* edge.

5.8.3.22 void sf_hal_gpio_powerOff (void)

Function for *enabling* power supply of the transceiver.

5.8.3.23 void sf_hal_gpio_powerOn (void)

Pulling down the MCU pin connected to the shutdown pin (SDN) will power down the Si446x transceiver. The MCU pin is specified by the appropriate configuration (have a look at `/src/configs/`) using the RF_GPIO0_PORT and RF_SDN_PIN macros.

Function for *disabling* power supply of the transceiver.

5.9 Si446x driver SPI HAL

5.9.1 Detailed Description

The SPI HAL module provides access to the SPI which is needed to configure and exchange data with Si446x transceivers.

Furthermore, the RF driver also requires to have the chip select handling implemented. This is chip select might be part of the MCU SPI periphery or might be independently configured GPIO. This is very dependent on the interconnection betweeen MCU and transceiver.

The SPI access is implemented in `/src/target/`sf_hal_spi.c.

5.9.2 Data Structure Documentation

5.9.2.1 struct s_spi_txTx_t

Structure to save rx or tx parameters.

Data Fields

uint16_t	i_len	Number of bytes to send or receive.
uint8_t *	pc_data	Temporary data storage.
uint8_t *	pc_end	End address of writing or reading.

5.9.3 Macro Definition Documentation

5.9.3.1 #define SPI_ISR_TIMEOUT 1000U

Stack includes

EFM32LIB includes

Timeout to prevent blocking errors in retries.

5.9.4 Typedef Documentation

5.9.4.1 typedef void(* fp_hal_spi_event)(uint16_t i_len)

Callback function on transmission and reception to notify RF module. Will be called each time the transmission of previously delivered data has finished.

Parameters

i_len	Length of the transmitted data.

5.9.5 Function Documentation

5.9.5.1 void sf_hal_spi_chipDeselect (void)

Will be called by the RF driver for *disabling* the chip select pin.

5.9.5.2 void sf_hal_spi_chipSelect (void)

Will be called by the RF driver in order to *enable* the chip select pin.

5.9.5.3 void sf_hal_spi_init (**fp_hal_spi_event** fp_rx, **fp_hal_spi_event** fp_tx)

Initialization of the SPI interface. This function will be called by the RF driver for initializing the SPI interface of the MCU as well as registering the callback pointers for signalling SPI Tx and Rx interrupts to the RF driver.

sf_hal_spi_init()

Parameters

fp_rx	Callback pointer for signalling *Rx* interrupts at the SPI.
fp_tx	Callback pointer for signalling *Tx* interrupts at the SPI.

Configure the SPI interface of the MCU

5.9.5.4 void sf_hal_spi_xfer (uint8_t * pc_dataWrite, uint8_t * pc_dataRead, uint16_t i_len)

Exchanges multiple bytes interrupt driven via SPI . This functions handles a data exchange operation at the SPI. It'll transmit *i_len* bytes located at address *pc_dataWrite* and simultaneously stores *i_len* bytes received at address location *pc_dataRead*. This function is non-blocking, uses ISRs for handling the exchange and is meant to handle many data bytes.

Parameters

pc_dataWrite	Pointer to the data to be transmitted by writing to SPI.
pc_dataRead	Pointer where to store received data.

i_len	Number of bytes to be transmitted or received.

5.9.5.5 uint8_t sf_hal_spi_xferBlock (uint8_t * pc_dataWrite, uint8_t * pc_dataRead, uint16_t i_len)

Exchanges multiple data bytes via SPI. Compared to sf_hal_spi_xfer this function does the very same, except of it is blocking during transmission. This type of SPI data exchange is meant for exchanging just a few bytes when ISR handling would cause too much overhead, e.g. when reading a single register from the transceiver.

Parameters

pc_dataWrite	Pointer to the data to be transmitted by writing to SPI.
pc_dataRead	Pointer where to store received data.
i_len	Number of bytes to be transmitted or received.

Returns

TX: Status byte. RX: Always 0.

5.9.5.6 void SPI_USART_TXIRQ_HANDLER_FNC (void)

disable Tx interrupts

5.9.6 Variable Documentation

5.9.6.1 volatile uint16_t gi_spi_isr_wtd

Watchdog ISR timer.

5.9.6.2 USART_TypeDef* gps_spi

SPI instance

5.9.6.3 volatile **s_spi_txTx_t** gs_spi_rx_tx

Structure to save rx or tx parameters.

5.10 Si446x driver MCU HAL

5.10.1 Detailed Description

This module is required to provide some MCU core related functionality to the RF driver.

5.10.2 Function Documentation

5.10.2.1 uint32_t sf_hal_mcu_getClockSpeed (void)

Function for retrieving the MCU clock speed. This function is required by the RF driver in order to implement tiny delays, which are necessar for proper communication with the attached transceiver.
Note

> Please note, this is the only MCU related hardware abstracting function required by the Si446x driver module and has been implemented in `/src/target/`sf_hal_mcu.c

Chapter 6

File Documentation

6.1 D:/Development/wmbus-customers-silabs/src/hal/rf/sf_hal_gpio.h File Reference

HAL for GPIO implementations.

Typedefs

- typedef void(* fp_hal_gpio_radioIsr)(void)

Functions

- void sf_hal_gpio_powerOff (void)
- void sf_hal_gpio_powerOn (void)
- void sf_hal_gpio_initGPIOx (void)
- void sf_hal_gpio_irqRisingEdgeGPIO0 (void)
- void sf_hal_gpio_irqFallingEdgeGPIO0 (void)
- void sf_hal_gpio_irqEnableGPIO0 (void)
- void sf_hal_gpio_irqDisableGPIO0 (void)
- void sf_hal_gpio_irqClearFlagGPIO0 (void)
- bool_t sf_hal_gpio_irqFlagIsSetGPIO0 (void)
- void sf_hal_gpio_modeRxSetGPIO0 (void)
- void sf_hal_gpio_modeTxSetGPIO0 (void)
- bool_t sf_hal_gpio_isHighGPIO0 (void)
- bool_t sf_hal_gpio_isLowGPIO0 (void)
- void sf_hal_gpio_irqRisingEdgeGPIO1 (void)
- void sf_hal_gpio_irqFallingEdgeGPIO1 (void)
- void sf_hal_gpio_irqEnableGPIO1 (void)

- void sf_hal_gpio_irqDisableGPIO1 (void)
- void sf_hal_gpio_irqClearFlagGPIO1 (void)
- bool_t sf_hal_gpio_irqFlagIsSetGPIO1 (void)
- void sf_hal_gpio_modeRxSetGPIO1 (void)
- void sf_hal_gpio_modeTxSetGPIO1 (void)
- bool_t sf_hal_gpio_isHighGPIO1 (void)
- bool_t sf_hal_gpio_isLowGPIO1 (void)

6.1.1 Detailed Description

HAL for GPIO implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.2 D:/Development/wmbus-customers-silabs/src/hal/rf/sf_hal_spi.h File Reference

HAL module implementing the SPI interface for the RF driver.

Typedefs

- typedef void(* fp_hal_spi_event)(uint16_t i_len)

  Callback function on transmission and reception to notify RF module. Will be called each time the transmission of previously delivered data has finished.

Functions

- void sf_hal_spi_init (fp_hal_spi_event fp_rx, fp_hal_spi_event fp_tx)

  Initialization of the SPI interface. This function will be called by the RF driver for initializing the SPI interface of the MCU as well as registering the callback pointers for signalling SPI Tx and Rx interrupts to the RF driver.
- void sf_hal_spi_chipSelect (void)
- void sf_hal_spi_chipDeselect (void)

- void sf_hal_spi_xfer (uint8_t *pc_dataWrite, uint8_t *pc_dataRead, uint16_t i_len)

  Exchanges multiple bytes interrupt driven via SPI . This functions handles a data exchange operation at the SPI. It'll transmit i_len bytes located at address pc_dataWrite and simultaneously stores i_len bytes received at address location pc_dataRead. This function is non-blocking, uses ISRs for handling the exchange and is meant to handle many data bytes.

- uint8_t sf_hal_spi_xferBlock (uint8_t *pc_dataWrite, uint8_t *pc_dataRead, uint16_t i_len)

  Exchanges multiple data bytes via SPI. Compared to sf_hal_spi_xfer this function does the very same, except of it is blocking during transmission. This type of SPI data exchange is meant for exchanging just a few bytes when ISR handling would cause too much overhead, e.g. when reading a single register from the transceiver.

6.2.1 Detailed Description

HAL module implementing the SPI interface for the RF driver.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.3 D:/Development/wmbus-customers-silabs/src/hal/rf/sf_rf_hal_mcu.h File Reference

HAL for MCU implementations needed by RF driver.

Functions

- uint32_t sf_hal_mcu_getClockSpeed (void)

  Function for retrieving the MCU clock speed. This function is required by the RF driver in order to implement tiny delays, which are necessar for proper communication with the attached transceiver.

6.3.1 Detailed Description

HAL for MCU implementations needed by RF driver.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.4 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal.h File Reference

Hardware Abstraction Layer Application Programming Interface.

Enumerations

- enum E_HAL_STATUS_t {
  E_HAL_STATUS_INVALID, E_HAL_STATUS_SUCCESS, E_HAL_STATUS_MCU_ERROR, E_HAL_STATUS_MEM_ERROR,
  E_HAL_STATUS_RF_ERROR, E_HAL_STATUS_AES_ERROR, E_HAL_STATUS_UPDATE_ERROR, E_HAL_STATUS_TMR_ERROR,
  E_HAL_STATUS_UART_ERROR, E_HAL_STATUS_UNKNOWN_ERROR }

Functions

- E_HAL_STATUS_t wmbus_hal_init (void)

  Initializes all hardware components.

6.4.1 Detailed Description

Hardware Abstraction Layer Application Programming Interface.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.5 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal_aes.h File Reference

HAL module for hardware accelerated AES en-/decryption.

Functions

- bool_t wmbus_hal_aes_init (void)

  Performs the initialization of the AES HAL module. This function usually will be called by wmbus_hal_init() and enables the AES HAL module to initialize the hardware.

- bool_t wmbus_hal_aes_setKey (const uint8_t *pc_key)

  Sets key for en-/decryption. Sets key to be used for any encryption or decryption procedure by the AES HAL module. Basically the key will be maintained by the stack and given to the AES HAL module for maintaing the underlying hardware. There is no need to call this function by user application.

- bool_t wmbus_hal_aes_encrypt (const uint8_t *pc_in, uint8_t *pc_out)

  Encrypts a single block only. Encrypts a single block only, using the key that is currently stored within the HAL module. Block length for AES is fixed to 16 bytes.

- bool_t wmbus_hal_aes_decrypt (const uint8_t *pc_in, uint8_t *pc_out)

  Decrypts a single block only. Decrypts a single block only, using the key that is currently stored within the HAL module. Block length for AES is fixed to 16 bytes.

- bool_t wmbus_hal_aes_cbcEncrypt (const uint8_t *pc_in, uint8_t *pc_out, uint16_t n_block, uint8_t *pc_iv)

  Performs CBC encryption on a given number of blocks. This function performs CBC encryption on a given number of blocks, using the given initialization vector (iv).

- bool_t wmbus_hal_aes_cbcDecrypt (const uint8_t *pc_in, uint8_t *pc_out, uint16_t n_block, uint8_t *pc_iv)

  Performs CBC decryption on a given number of blocks. This function performs CBC decryption on a given number of blocks, using the given initialization vector (iv).

- bool_t wmbus_hal_aes_ctrEncrypt (const uint8_t *pc_in, uint8_t *pc_out, uint8_t c_len, uint8_t *pc_iv)

  Performs CTR encryption on a given number of blocks. This function performs CTR encryption on a given number of blocks, using the given initialization vector (iv).

- bool_t wmbus_hal_aes_ctrDecrypt (const uint8_t *pc_in, uint8_t *pc_out, uint8_t c_len, uint8_t *pc_iv)

  Performs CTR decryption on a given number of blocks. This function performs CTR decryption on a given number of blocks, using the given initialization vector (iv).

6.5.1 Detailed Description

HAL module for hardware accelerated AES en-/decryption.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.6 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal_mcu.h File Reference

HAL module for MCU core related functions.

Functions

- bool_t wmbus_hal_mcu_init (void)

  Initializes the MCU core. In case the MCU core needs some kind of initialization (e.g. setting up the core clock, ...), this will be handled within the MCU initialization.

- void wmbus_hal_mcu_reset (void)

  Resets the MCU. This function handles the proper reset/restart of the complete MCU.

6.6.1 Detailed Description

HAL module for MCU core related functions.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.7 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal_mem.h File Reference

Hardware Abstraction Layer Application Programming Interface for memory implementations.

Functions

- bool_t wmbus_hal_mem_init (void)

  Initializes the memory module, in case the non-volatile needs any preparation before using it.

- uint16_t wmbus_hal_mem_write (uint8_t *pc_data, uint16_t i_len, uint32_t l_offset)

  Writes data into the non-volatile memory. This function provides access to the non-volatile memory for storing stack internal information that need to be kept in case of power loss.

- uint16_t wmbus_hal_mem_read (uint8_t *pc_data, uint16_t i_len, uint32_t l_offset)

  Reads data from the non-volatile memory. Especially on startup of the device, the stack needs to read importand data that has been saved earlier (e.g. device address).

6.7.1 Detailed Description

Hardware Abstraction Layer Application Programming Interface for memory implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.8 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal_rf.h File Reference

Hardware Abstraction Layer Application Programming Interface for radio implementations.

Macros

- #define HAL_RF_QUALITY_LEN 2U
- #define HAL_RF_NEW_TLG 0xFFFFU
- #define HAL_RF_MODE_C_PREAMBLE_FIRST 0x54U
- #define HAL_RF_MODE_C_FRAMETYPE_A 0xCDU
- #define HAL_RF_MODE_C_FRAMETYPE_B 0x3DU
- #define HAL_RF_MODE_N_SYNC1 0xF6U
- #define HAL_RF_MODE_N_SYNC2_FRAME_A 0x8DU
- #define HAL_RF_MODE_N_SYNC2_FRAME_B 0x72U

Typedefs

- typedef void(* fp_hal_rf_evt_tx)(uint16_t i_len)

Callback which indicates a finished transmission.

- typedef void(* fp_hal_rf_evt_rx)(uint16_t i_len, E_WMBUS_FRAME_t e_frameType)

  Callback which indicates a new reception.

Enumerations

- enum E_HAL_RF_POWERMODE_t { E_HAL_RF_POWERMODE_OFF, E_HAL_RF_POWERMODE_IDLE, E_HAL_RF_POWERMODE_RX, E_HAL_RF_POWERMODE_MAX }
- enum E_HAL_RF_MODE_t { E_HAL_RF_MODE_RUN, E_HAL_RF_MODE_WAIT }
- enum E_HAL_RF_CALIBRATE_t { E_HAL_RF_CALIB_CALIBRATE_STORE, E_HAL_RF_CALIB_LOAD, E_HAL_RF_CALIB_OFF }
- enum E_HAL_RF_POSTAMBLE_t { E_HAL_RF_POSTAMBLE_NONE = 0x00, E_HAL_RF_POSTAMBLE_ODD = 0x55, E_HAL_RF_POSTAMBLE_EVEN = 0xAA }
- enum E_HAL_RF_CS_STATUS_t { E_HAL_RF_CS_STATUS_NO_CARRIER_DETECTED, E_HAL_RF_CS_STATUS_CARRIER_DETECTED, E_HAL_RF_CS_STATUS_INVALID_STATE, E_HAL_RF_CS_STATUS_ERROR }

Functions

- bool_t wmbus_hal_rf_init (void)

  Initializes the RF module. This function simply initializes the RF driver itself. Before any communication starts, the stack will call wmbus_hal_rf_start.

- bool_t wmbus_hal_rf_setCallback (fp_hal_rf_evt_tx fp_tx, fp_hal_rf_evt_rx fp_rx)

  Sets the callback pointer for TX and RX events.

- void wmbus_hal_rf_powerOff (void)

  Turns off the supply voltage of the RF module.

- void wmbus_hal_rf_powerOn (void)

  Turns on the supply voltage of the RF module.

- void wmbus_hal_rf_start (void)

  This function will be called once after the initialisation to start operation of the RF driver.

- bool_t wmbus_hal_rf_txInit (uint16_t i_len, E_WMBUS_FRAME_t e_frameType, E_WMBUS_MODE_t e_mode)

  Initiates the transmission of data. This function will be called for initialising the transmission procedure.

- bool_t wmbus_hal_rf_txData (uint8_t *pc_data, uint16_t i_len)

Transmits a chunk of data. For Wireless M-Bus the data will be sent chunk wise. Therefore all data to be transmitted will be delivered to the RF driver by sequentially calling this function.

- bool_t wmbus_hal_rf_txFinish (void)

  Finalises the transmission of data. This function will be called by the stack to finalise a transmission, independently if transmission succeeded of failed. At least wmbus_rf_txInit should be called before first.

- bool_t wmbus_hal_rf_rxInit (uint8_t *pc_quality, uint8_t c_len)

  Initiates the reception of data. As soon as the stack has been informed about the detection of a telegram, the stack will initialise the reception procedure by calling this function.

- bool_t wmbus_hal_rf_rxData (uint8_t *pc_data, uint16_t i_len)

  Retrieves a chunk of data from the RF driver. Just like the transmission, reception of data is done chunk-wise. For this, the function will be called as fast as possible to retrieve all data received by the transceiver.

- bool_t wmbus_hal_rf_rxFinish (E_HAL_RF_MODE_t e_mode)

  Finalises the reception of data. At least wmbus_rf_rxInit should be called before.

- bool_t wmbus_hal_rf_reset (E_HAL_RF_CALIBRATE_t e_calibrate)

  Resets the RF module. The stack will request the RF driver to reset any time the stack assumes the RF driver needs to re-enter a defined state.

- bool_t wmbus_hal_rf_setRfChannel (uint16_t i_channel)

  Sets the channel to use for RF communication.

- void wmbus_hal_rf_txSetPostamble (E_HAL_RF_POSTAMBLE_t e_postamble)

  Sets the postamble to be used for transmissions.

- uint16_t wmbus_hal_rf_getRfChannel (void)

  Returns the currently channel of the RF communication.

- bool_t wmbus_hal_rf_setPowerMode (E_HAL_RF_POWERMODE_t e_powermode)

  Sets the power mode of the RF driver.

- bool_t wmbus_hal_rf_setSignalStrength (uint8_t c_signal)

  Sets the signal strength for transmission. Requests the RF driver to set the output power for the transceiver to the appropriate value.

- bool_t wmbus_hal_rf_setFrequencyOffset (sint16_t si_freqOffset)

  Sets the frequency offset of the carrier.

- uint8_t wmbus_hal_rf_getSignalStrength (void)

  Get the signal strength of the transceiver. Requests the RF driver to read the currently configured output power.

- void wmbus_hal_rf_sleep (void)

  Requests the RF driver to enter sleep mode.

- void wmbus_hal_rf_wake (void)

  Requests the RF driver to wake up.

- void wmbus_hal_rf_run (void)

  Main loop computing time for the RF driver. The stack calls this function to forward computing time earned by the main loop to the RF driver. Commonly RF driver do not need generic computing time, but in case they need, this would be the entry point.

- uint16_t wmbus_hal_rf_getTelegramDelay (void)

  Retrieves the delay of receiving a telegram. Will be called for retrieving the delay in 500us steps the transceiver needs for receiving and forcing the data into the stack.

- E_HAL_RF_CS_STATUS_t wmbus_hal_rf_carrierSense (sint8_t c_rssiThres)

  Listens to the medium and reports whether its free or not. The functions uses the chip internal features to perform carrier sense. Before calling this function, the chip itself and the SPI communication interface must be initialized. The function sets the chip in the correct state to perform carrier sense.

- bool_t wmbus_hal_rf_setDataRate (E_WMBUS_DATA_RATE_t e_dataRate)

  Sets the data rate of the transceiver. Requests the RF driver to configure the transceiver for the requested data rate on transmission and receptions.

- bool_t wmbus_hal_rf_setRxSenseTuning (E_WMBUS_MODE_t e_mode)

  Enables mode C collector RX configuration for receiving mode T and C. Please note that this feature is optional as it is limited to be used for mode C collectors only. Additionally, this feature must be supported by the underlying RF driver. You may ask your support contact for further information on this feature. The current tuning configuration can be requested by using wmbus_hal_rf_getRxSenseTuning.

- E_WMBUS_MODE_t wmbus_hal_rf_getRxSenseTuning (void)

  Returns the current transceiver RX configuration. Returns the current transceiver RX configuration whether it is tuned for reception of either mode T or mode C meters. Please note that this feature is optional as it is limited to be used for mode C collectors only. Additionally, this feature must be supported by the underlying RF driver. You may ask your support contact for further information on this feature. The tuning configuration can be set by using wmbus_hal_rf_setRxSenseTuning.

6.8.1 Detailed Description

Hardware Abstraction Layer Application Programming Interface for radio implementations.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, `http://www.stackforce.de`

Author

STACKFORCE

6.9 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal_tmr.h File Reference

Hardware Abstraction Layer Application Programming Interface for timer implementations.

Typedefs

- typedef void(* fp_hal_tmr_cb)(void)

  Callback which indicates a finished transmission.

Functions

- bool_t wmbus_hal_tmr_init (uint16_t i_ticksPerSecond)

  Initializes the timer. Requests the driver for the hardware timer to initialize for timer interrupt at an interval of the given value.

- bool_t wmbus_hal_tmr_setCallback (fp_hal_tmr_cb fp_tmr)

  Sets the callback for the timer event. Provides a pointer to the function to be called, when the hardware timer is causing an interrupt due to the configured interval.

- void wmbus_hal_tmr_enable (void)

  Enables the timer interrupts. As soon as the stack wants to be informed about timer ticks, it'll enable the hardware timer interrupts by calling this function.

- void wmbus_hal_tmr_disable (void)

  Disables the timer interrupts. The very same like wmbus_hal_tmr_enable, but the other way around.

- bool_t wmbus_hal_tmr_set (uint16_t ui_counterValue)

  Sets the timer tick counter. For the dynamic timer function, the stack may request the hardware timer driver to set the tick counter to the requested value.

- bool_t wmbus_hal_tmr_offset (sint16_t si_offset)

  Adds a positive or negative offset to the timer tick counter.

6.9.1 Detailed Description

Hardware Abstraction Layer Application Programming Interface for timer implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.10 D:/Development/wmbus-customers-silabs/src/stack/inc/pub/hal/wmbus_hal_uart.h File Reference

Hardware Abstraction Layer Application Programming Interface for UART implementations.

Enumerations

- enum E_HAL_UART_PARITY_t { E_HAL_UART_PARITY_NONE, E_HAL_UART_PARITY_EVEN, E_HAL_UART_PARITY_ODD }
- enum E_HAL_UART_DATABITS_t { E_HAL_UART_DATABITS_8, E_HAL_UART_DATABITS_7 }
- enum E_HAL_UART_STOPBITS_t { E_HAL_UART_STOPBITS_0, E_HAL_UART_STOPBITS_1, E_HAL_UART_STOPBITS_15, E_HAL_UART_STOPBITS_2 }

Functions

- bool_t wmbus_hal_uart_init (void)

  Initializes the UART. Within this function commonly the underlying UART interface will be configured accordingly.

- uint16_t wmbus_hal_uart_write (uint8_t *pc_data, uint16_t i_len)

  Writes data to the UART. This function will be called by the serial layer of the stack when stack requests to send data to the connected host.

- uint16_t wmbus_hal_uart_read (uint8_t *pc_data, uint16_t i_len)

  Reads data from the UART. This function retrieves the requested amount of data from the UART driver.

- uint16_t wmbus_hal_uart_cntRxBytes (void)

  Returns the number of ready-to-be-fetched bytes. This function is called by the stack in preparation to a receive operation and is for reading the number of received bytes, that are buffered by the stack.

- uint16_t wmbus_hal_uart_cntTxBytes (void)

  Returns the current capability of transmitting bytes. In preparation to a transmit operation, the stack may request the number of bytes that could be transmitted. The UART driver shall return the number of bytes that possibly could be buffered until real transmission.

- bool_t wmbus_hal_uart_isRxOverflow (void)

Checks if there was a RX buffer overflow. This function will be called by the stack to double check if bytes have been lost, e.g. in case the stack has been receiving not plausible data. The UART driver should use this function to also reset its RX overflow error.

- void wmbus_hal_uart_com_TxFinish (void)

  Signals that a serial telegram is complete. This function will be called by the serial layer, as soon as a complete serial telegram has been sent using the wmbus_hal_uart_write function.

- void wmbus_hal_uart_com_run (void)

  Run the serial protocol state machine. In case the UART driver needs to maintain a state machine, the stack will provide main loop computing time by calling this function.

6.10.1 Detailed Description

Hardware Abstraction Layer Application Programming Interface for UART implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.11 D:/Development/wmbus-customers-silabs/src/target/sf_hal.c File Reference

Core HAL module.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\utils\wmbus_timer_api.h"
#include "inc\pub\utils\wmbus_tlg_api.h"
#include "inc\pub\utils\wmbus_api.h"
#include "inc\pub\hal\wmbus_hal.h"
#include "inc\pub\hal\wmbus_hal_mcu.h"
#include "inc\pub\hal\wmbus_hal_mem.h"
#include "inc\pub\hal\wmbus_hal_tmr.h"
#include "inc\pub\hal\wmbus_hal_uart.h"
#include "inc\pub\hal\wmbus_hal_aes.h"
#include "inc\pub\hal\wmbus_hal_rf.h"
```

Macros

- #define HAL_STATUS_OK(A) (A == E_HAL_STATUS_SUCCESS)

Functions

- E_HAL_STATUS_t wmbus_hal_init (void)

  Initializes all hardware components.

6.11.1 Detailed Description

Core HAL module.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.12 D:/Development/wmbus-customers-silabs/src/target/sf_hal_aes.c File Reference

HAL for AES implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "em_cmu.h"
#include "em_aes.h"
```

Macros

- #define EFM_AES_KEY_LEN (16U)
- #define EFM_AES_BLOCK_LEN (16U)

Functions

- bool_t wmbus_hal_aes_init (void)

Performs the initialization of the AES HAL module. This function usually will be called by wmbus_hal_init() and enables the AES HAL module to initialize the hardware.

- bool_t wmbus_hal_aes_setKey (const uint8_t *pc_key)

  Sets key for en-/decryption. Sets key to be used for any encryption or decryption procedure by the AES HAL module. Basically the key will be maintained by the stack and given to the AES HAL module for maintaing the underlying hardware. There is no need to call this function by user application.

- bool_t wmbus_hal_aes_cbcDecrypt (const uint8_t *pc_in, uint8_t *pc_out, uint16_t n_block, uint8_t *pc_iv)

  Performs CBC decryption on a given number of blocks. This function performs CBC decryption on a given number of blocks, using the given initialization vector (iv).

- bool_t wmbus_hal_aes_cbcEncrypt (const uint8_t *pc_in, uint8_t *pc_out, uint16_t n_block, uint8_t *pc_iv)

  Performs CBC encryption on a given number of blocks. This function performs CBC encryption on a given number of blocks, using the given initialization vector (iv).

- bool_t wmbus_hal_aes_encrypt (const uint8_t *pc_in, uint8_t *pc_out)

  Encrypts a single block only. Encrypts a single block only, using the key that is currently stored within the HAL module. Block length for AES is fixed to 16 bytes.

6.12.1 Detailed Description

HAL for AES implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.13 D:/Development/wmbus-customers-silabs/src/target/sf_hal_gpio.c File Reference

HAL for GPIO implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "sf_hal_gpio.h"
#include "em_gpio.h"
```

Macros

- #define **RF_GPIO0_FLAG** (1 << RF_GPIO0_PIN)
- #define **RF_GPIO1_FLAG** (1 << RF_GPIO1_PIN)

Functions

- void sf_hal_gpio_powerOn (void)
- void sf_hal_gpio_powerOff (void)
- void sf_hal_gpio_initGPIOx (void)
- void sf_hal_gpio_irqRisingEdgeGPIO0 (void)
- void sf_hal_gpio_irqFallingEdgeGPIO0 (void)
- void sf_hal_gpio_irqEnableGPIO0 (void)
- void sf_hal_gpio_irqDisableGPIO0 (void)
- void sf_hal_gpio_irqClearFlagGPIO0 (void)
- bool_t sf_hal_gpio_irqFlagIsSetGPIO0 (void)
- void sf_hal_gpio_modeRxSetGPIO0 (void)
- void sf_hal_gpio_modeTxSetGPIO0 (void)
- bool_t sf_hal_gpio_isHighGPIO0 (void)
- bool_t sf_hal_gpio_isLowGPIO0 (void)
- void sf_hal_gpio_irqRisingEdgeGPIO1 (void)
- void sf_hal_gpio_irqFallingEdgeGPIO1 (void)
- void sf_hal_gpio_irqEnableGPIO1 (void)
- void sf_hal_gpio_irqDisableGPIO1 (void)
- void sf_hal_gpio_irqClearFlagGPIO1 (void)
- bool_t sf_hal_gpio_irqFlagIsSetGPIO1 (void)
- void sf_hal_gpio_modeRxSetGPIO1 (void)
- void sf_hal_gpio_modeTxSetGPIO1 (void)
- bool_t sf_hal_gpio_isHighGPIO1 (void)
- bool_t sf_hal_gpio_isLowGPIO1 (void)

6.13.1 Detailed Description

HAL for GPIO implementations.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.14 D:/Development/wmbus-customers-silabs/src/target/sf_hal_leuart.c File Reference

HAL for LEUART implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\hal\wmbus_hal_uart.h"
#include "em_chip.h"
#include "em_emu.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_leuart.h"
```

Macros

- #define UART_BUFFER_RX_LEN 20U
- #define UART_BUFFER_TX_LEN 50U
- #define UART_USE_TX_IRQ TRUE

Functions

- void loc_sf_uart_rx_isr (void)
- void loc_sf_uart_tx_isr (void)
- bool_t wmbus_hal_uart_init (void)

  Initializes the UART. Within this function commonly the underlying UART interface will be configured accordingly.

- uint16_t wmbus_hal_uart_write (uint8_t *pc_data, uint16_t i_len)

  Writes data to the UART. This function will be called by the serial layer of the stack when stack requests to send data to the connected host.

- uint16_t wmbus_hal_uart_read (uint8_t *pc_data, uint16_t i_len)

Reads data from the UART. This function retrieves the requested amount of data from the UART driver.

- uint16_t wmbus_hal_uart_cntRxBytes (void)

  Returns the number of ready-to-be-fetched bytes. This function is called by the stack in preparation to a receive operation and is for reading the number of received bytes, that are buffered by the stack.

- uint16_t wmbus_hal_uart_cntTxBytes (void)

  Returns the current capability of transmitting bytes. In preparation to a transmit operation, the stack may request the number of bytes that could be transmitted. The UART driver shall return the number of bytes that possibly could be buffered until real transmission.

- bool_t wmbus_hal_uart_isRxOverflow (void)

  Checks if there was a RX buffer overflow. This function will be called by the stack to double check if bytes have been lost, e.g. in case the stack has been receiving not plausible data. The UART driver should use this function to also reset its RX overflow error.

- void wmbus_hal_uart_com_TxFinish (void)

  Signals that a serial telegram is complete. This function will be called by the serial layer, as soon as a complete serial telegram has been sent using the wmbus_hal_uart_write function.

- void wmbus_hal_uart_com_run (void)

  Run the serial protocol state machine. In case the UART driver needs to maintain a state machine, the stack will provide main loop computing time by calling this function.

- void **LEUART0_IRQHandler** (void)

Variables

- LEUART_TypeDef * **usart_interface** = UART_USART_INTERFACE
- LEUART_Init_TypeDef **usart_settings**
- volatile uint8_t gc_uart_bufferRx [UART_BUFFER_RX_LEN]
- volatile uint8_t * gpc_uart_bufferRxWrite
- volatile uint8_t * gpc_uart_bufferRxRead
- volatile uint16_t gi_uart_bufferRxLen
- volatile bool_t gb_uart_bufferRxOverflow
- volatile uint8_t gc_uart_bufferTx [UART_BUFFER_TX_LEN]
- volatile uint8_t * gpc_uart_bufferTxWrite
- volatile uint8_t * gpc_uart_bufferTxRead
- volatile uint16_t gi_uart_bufferTxLen

6.14.1 Detailed Description

HAL for LEUART implementations.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.15 D:/Development/wmbus-customers-silabs/src/target/sf_hal_mcu.c File Reference

HAL for implementations relevant to the MCU core.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_usb.h"
```

Functions

- bool_t wmbus_hal_mcu_init (void)

  Initializes the MCU core. In case the MCU core needs some kind of initialization (e.g. setting up the core clock, ...), this will be handled within the MCU initialization.

- void wmbus_hal_mcu_reset (void)

  Resets the MCU. This function handles the proper reset/restart of the complete MCU.

- uint32_t sf_hal_mcu_getClockSpeed (void)

  Function for retrieving the MCU clock speed. This function is required by the RF driver in order to implement tiny delays, which are necessar for proper communication with the attached transceiver.

6.15.1 Detailed Description

HAL for implementations relevant to the MCU core.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.16 D:/Development/wmbus-customers-silabs/src/target/sf_hal_mem.c File Reference

HAL for access to non-volatile memory.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\hal\wmbus_hal_mem.h"
#include "em_device.h"
#include "em_msc.h"
```

Macros

- #define MEM_FLASH_WRITE_RETRIES (3U)
- #define MEM_START_ADDR (FLASH_SIZE - FLASH_PAGE_SIZE)
- #define MEM_END_ADDR (MEM_START_ADDR + FLASH_PAGE_SIZE)

Functions

- uint16_t loc_datamemory_writeData (uint32_t l_addr, uint8_t *pc_data, uint16_t i_len)

  Handles writing of data. This function handles the write procedure to the flash.

- bool_t wmbus_hal_mem_init (void)

  Initializes the memory module, in case the non-volatile needs any preparation before using it.

- uint16_t wmbus_hal_mem_write (uint8_t *pc_data, uint16_t i_len, uint32_t l_offset)

  Writes data into the non-volatile memory. This function provides access to the non-volatile memory for storing stack internal information that need to be kept in case of power loss.

- uint16_t wmbus_hal_mem_read (uint8_t *pc_data, uint16_t i_len, uint32_t l_offset)

  Reads data from the non-volatile memory. Especially on startup of the device, the stack needs to read importand data that has been saved earlier (e.g. device address).

6.16.1 Detailed Description

HAL for access to non-volatile memory.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.17 D:/Development/wmbus-customers-silabs/src/target/sf_hal_rf.c File Reference

HAL for RF implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\utils\wmbus_tlg_api.h"
#include "inc\pub\utils\wmbus_api.h"
#include "inc\pub\hal\wmbus_hal_rf.h"
#include "sf_rf.h"
#include "sf_hal_gpio.h"
```

Functions

- bool_t wmbus_hal_rf_init (void)

  Initializes the RF module. This function simply initializes the RF driver itself. Before any communication starts, the stack will call wmbus_hal_rf_start.

- void wmbus_hal_rf_powerOff (void)

  Turns off the supply voltage of the RF module.

- void wmbus_hal_rf_powerOn (void)

  Turns on the supply voltage of the RF module.

- void wmbus_hal_rf_start (void)

  This function will be called once after the initialisation to start operation of the RF driver.

- bool_t wmbus_hal_rf_txInit (uint16_t i_len, E_WMBUS_FRAME_t e_frameType, E_WMBUS_MODE_t e_↩ mode)

  Initiates the transmission of data. This function will be called for initialising the transmission procedure.

- bool_t wmbus_hal_rf_txData (uint8_t *pc_data, uint16_t i_len)

  Transmits a chunk of data. For Wireless M-Bus the data will be sent chunk wise. Therefore all data to be transmitted will be delivered to the RF driver by sequentially calling this function.

- bool_t wmbus_hal_rf_txFinish (void)

  Finalises the transmission of data. This function will be called by the stack to finalise a transmission, independently if transmission succeeded of failed. At least wmbus_rf_txInit should be called before first.

- bool_t wmbus_hal_rf_rxInit (uint8_t *pc_quality, uint8_t c_len)

  Initiates the reception of data. As soon as the stack has been informed about the detection of a telegram, the stack will initialise the reception procedure by calling this function.

- bool_t wmbus_hal_rf_rxData (uint8_t *pc_data, uint16_t i_len)

  Retrieves a chunk of data from the RF driver. Just like the transmission, reception of data is done chunk-wise. For this, the function will be called as fast as possible to retrieve all data received by the transceiver.

- bool_t wmbus_hal_rf_rxFinish (E_HAL_RF_MODE_t e_mode)

  Finalises the reception of data. At least wmbus_rf_rxInit should be called before.

- bool_t wmbus_hal_rf_reset (E_HAL_RF_CALIBRATE_t e_calibrate)

  Resets the RF module. The stack will request the RF driver to reset any time the stack assumes the RF driver needs to re-enter a defined state.

- bool_t wmbus_hal_rf_setRfChannel (uint16_t i_channel)

  Sets the channel to use for RF communication.

- void wmbus_hal_rf_txSetPostamble (E_HAL_RF_POSTAMBLE_t e_postamble)

  Sets the postamble to be used for transmissions.

- uint16_t wmbus_hal_rf_getRfChannel (void)

  Returns the currently channel of the RF communication.

- bool_t wmbus_hal_rf_setPowerMode (E_HAL_RF_POWERMODE_t e_powermode)

  Sets the power mode of the RF driver.

- bool_t wmbus_hal_rf_setSignalStrength (uint8_t c_signal)

  Sets the signal strength for transmission. Requests the RF driver to set the output power for the transceiver to the appropriate value.

- uint8_t wmbus_hal_rf_getSignalStrength (void)

  Get the signal strength of the transceiver. Requests the RF driver to read the currently configured output power.

- void wmbus_hal_rf_sleep (void)

  Requests the RF driver to enter sleep mode.

- void wmbus_hal_rf_wake (void)

Requests the RF driver to wake up.

- void wmbus_hal_rf_run (void)

  Main loop computing time for the RF driver. The stack calls this function to forward computing time earned by the main loop to the RF driver. Commonly RF driver do not need generic computing time, but in case they need, this would be the entry point.

- uint16_t wmbus_hal_rf_getTelegramDelay (void)

  Retrieves the delay of receiving a telegram. Will be called for retrieving the delay in 500us steps the transceiver needs for receiving and forcing the data into the stack.

- bool_t wmbus_hal_rf_setCallback (fp_hal_rf_evt_tx fp_tx, fp_hal_rf_evt_rx fp_rx)

  Sets the callback pointer for TX and RX events.

- bool_t wmbus_hal_rf_setFrequencyOffset (sint16_t si_freqOffset)

  Sets the frequency offset of the carrier.

- void **sf_rf_evt_criticalError** (E_RF_ERROR_t e_err)
- void **GPIO_ODD_IRQHandler** (void)
- void **GPIO_EVEN_IRQHandler** (void)

6.17.1 Detailed Description

HAL for RF implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.18 D:/Development/wmbus-customers-silabs/src/target/sf_hal_spi.c File Reference

HAL for SPI implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\hal\wmbus_hal_mcu.h"
#include "sf_hal_spi.h"
#include "sf_rf_hal_mcu.h"
#include "em_gpio.h"
#include "em_cmu.h"
#include "em_usart.h"
```

Data Structures

- struct s_spi_txTx_t

Macros

- #define SPI_ISR_TIMEOUT 1000U

Functions

- void sf_hal_spi_init (fp_hal_spi_event fp_rx, fp_hal_spi_event fp_tx)

  Initialization of the SPI interface. This function will be called by the RF driver for initializing the SPI interface of the MCU as well as registering the callback pointers for signalling SPI Tx and Rx interrupts to the RF driver.

- void sf_hal_spi_chipSelect (void)
- void sf_hal_spi_chipDeselect (void)
- void sf_hal_spi_xfer (uint8_t *pc_dataWrite, uint8_t *pc_dataRead, uint16_t i_len)

  Exchanges multiple bytes interrupt driven via SPI . This functions handles a data exchange operation at the SPI. It'll transmit i_len bytes located at address pc_dataWrite and simultaneously stores i_len bytes received at address location pc_dataRead. This function is non-blocking, uses ISRs for handling the exchange and is meant to handle many data bytes.

- uint8_t sf_hal_spi_xferBlock (uint8_t *pc_dataWrite, uint8_t *pc_dataRead, uint16_t i_len)

  Exchanges multiple data bytes via SPI. Compared to sf_hal_spi_xfer this function does the very same, except of it is blocking during transmission. This type of SPI data exchange is meant for exchanging just a few bytes when ISR handling would cause too much overhead, e.g. when reading a single register from the transceiver.

- void **SPI_USART_RXIRQ_HANDLER_FNC** (void)
- void SPI_USART_TXIRQ_HANDLER_FNC (void)

Variables

- volatile s_spi_txTx_t gs_spi_rx_tx
- volatile uint16_t gi_spi_isr_wtd
- USART_TypeDef * gps_spi

6.18.1 Detailed Description

HAL for SPI implementations.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.19 D:/Development/wmbus-customers-silabs/src/target/sf_hal_tmr.c File Reference

HAL for TMR implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\hal\wmbus_hal_tmr.h"
#include "em_cmu.h"
#include "em_timer.h"
```

Macros

- #define TMR_DEFAULT

Functions

- bool_t wmbus_hal_tmr_init (uint16_t i_ticksPerSecond)

  Initializes the timer. Requests the driver for the hardware timer to initialize for timer interrupt at an interval of the given value.

- bool_t **wmbus_hal_tmr_deinit** (void)
- bool_t wmbus_hal_tmr_setCallback (fp_hal_tmr_cb fp_tmr)

  Sets the callback for the timer event. Provides a pointer to the function to be called, when the hardware timer is causing an interrupt due to the configured interval.

- void wmbus_hal_tmr_enable (void)

  Enables the timer interrupts. As soon as the stack wants to be informed about timer ticks, it'll enable the hardware timer interrupts by calling this function.

- void wmbus_hal_tmr_disable (void)

Disables the timer interrupts. The very same like wmbus_hal_tmr_enable, but the other way around.

- bool_t wmbus_hal_tmr_set (uint16_t ui_counterValue)

  Sets the timer tick counter. For the dynamic timer function, the stack may request the hardware timer driver to set the tick counter to the requested value.

- bool_t wmbus_hal_tmr_offset (sint16_t si_offset)

  Adds a positive or negative offset to the timer tick counter.

- void **TIMER1_IRQHandler** (void)

6.19.1 Detailed Description

HAL for TMR implementations.
Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de
Author

STACKFORCE

6.20 D:/Development/wmbus-customers-silabs/src/target/sf_hal_uart_Usb.c File Reference

HAL for UART USB implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\hal\wmbus_hal_uart.h"
#include "descriptors.h"
#include "em_chip.h"
#include "em_emu.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_usart.h"
#include "em_usb.h"
```

Macros

- #define USB_RX_BUF_SIZ USB_FS_BULK_EP_MAXSIZE

- #define **USB_TX_BUF_SIZ** USB_BUFFER_TX_LEN
- #define UART_BUFFER_RX_LEN 300U
- #define USB_BUFFER_TX_LEN 300U

Functions

- EFM32_PACK_START (1)
- **__attribute__** ((packed))
- **EFM32_PACK_END** ()
- bool_t wmbus_hal_uart_init (void)

  Initializes the UART. Within this function commonly the underlying UART interface will be configured accordingly.
- uint16_t wmbus_hal_uart_write (uint8_t *pc_data, uint16_t i_len)

  Writes data to the UART. This function will be called by the serial layer of the stack when stack requests to send data to the connected host.
- uint16_t wmbus_hal_uart_read (uint8_t *pc_data, uint16_t i_len)

  Reads data from the UART. This function retrieves the requested amount of data from the UART driver.
- uint16_t wmbus_hal_uart_cntRxBytes (void)

  Returns the number of ready-to-be-fetched bytes. This function is called by the stack in preparation to a receive operation and is for reading the number of received bytes, that are buffered by the stack.
- uint16_t wmbus_hal_uart_cntTxBytes (void)

  Returns the current capability of transmitting bytes. In preparation to a transmit operation, the stack may request the number of bytes that could be transmitted. The UART driver shall return the number of bytes that possibly could be buffered until real transmission.
- bool_t wmbus_hal_uart_isRxOverflow (void)

  Checks if there was a RX buffer overflow. This function will be called by the stack to double check if bytes have been lost, e.g. in case the stack has been receiving not plausible data. The UART driver should use this function to also reset its RX overflow error.
- void wmbus_hal_uart_com_TxFinish (void)

  Signals that a serial telegram is complete. This function will be called by the serial layer, as soon as a complete serial telegram has been sent using the wmbus_hal_uart_write function.
- void wmbus_hal_uart_com_run (void)

  Run the serial protocol state machine. In case the UART driver needs to maintain a state machine, the stack will provide main loop computing time by calling this function.

6.20.1 Detailed Description

HAL for UART USB implementations.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

6.21 D:/Development/wmbus-customers-silabs/src/target/sf_hal_usart↩ Usb.c File Reference

HAL for USART USB implementations.

```
#include "inc\pub\utils\wmbus_typedefs.h"
#include "inc\pub\hal\wmbus_hal_uart.h"
#include "descriptors.h"
#include "em_chip.h"
#include "em_emu.h"
#include "em_cmu.h"
#include "em_gpio.h"
#include "em_usart.h"
#include "em_usb.h"
```

Macros

- #define USB_RX_BUF_SIZ USB_FS_BULK_EP_MAXSIZE
- #define **USB_TX_BUF_SIZ** USB_BUFFER_TX_LEN
- #define UART_BUFFER_RX_LEN 300U
- #define USB_BUFFER_TX_LEN 300U

Functions

- EFM32_PACK_START (1)
- **__attribute__** ((packed))
- **EFM32_PACK_END** ()

- bool_t wmbus_hal_uart_init (void)

  Initializes the UART. Within this function commonly the underlying UART interface will be configured accordingly.

- uint16_t wmbus_hal_uart_write (uint8_t *pc_data, uint16_t i_len)

  Writes data to the UART. This function will be called by the serial layer of the stack when stack requests to send data to the connected host.

- uint16_t wmbus_hal_uart_read (uint8_t *pc_data, uint16_t i_len)

  Reads data from the UART. This function retrieves the requested amount of data from the UART driver.

- uint16_t wmbus_hal_uart_cntRxBytes (void)

  Returns the number of ready-to-be-fetched bytes. This function is called by the stack in preparation to a receive operation and is for reading the number of received bytes, that are buffered by the stack.

- uint16_t wmbus_hal_uart_cntTxBytes (void)

  Returns the current capability of transmitting bytes. In preparation to a transmit operation, the stack may request the number of bytes that could be transmitted. The UART driver shall return the number of bytes that possibly could be buffered until real transmission.

- bool_t wmbus_hal_uart_isRxOverflow (void)

  Checks if there was a RX buffer overflow. This function will be called by the stack to double check if bytes have been lost, e.g. in case the stack has been receiving not plausible data. The UART driver should use this function to also reset its RX overflow error.

- void wmbus_hal_uart_com_TxFinish (void)

  Signals that a serial telegram is complete. This function will be called by the serial layer, as soon as a complete serial telegram has been sent using the wmbus_hal_uart_write function.

- void wmbus_hal_uart_com_run (void)

  Run the serial protocol state machine. In case the UART driver needs to maintain a state machine, the stack will provide main loop computing time by calling this function.

6.21.1 Detailed Description

HAL for USART USB implementations.

Copyright

STACKFORCE GmbH, Heitersheim, Germany, http://www.stackforce.de

Author

STACKFORCE

Chapter 7

Contact information

In case of questions, requests for quotations or ideas for further improvements, please contact:

STACKFORCE GmbH
Poststrasse 35
79423 Heitersheim
Germany

Freiburg HRB 711613
Geschäftsführer: David Rahusen

Tel: +49 7634-69960-20
Fax: +49 7634-69960-30
Url: http://www.stackforce.de
E-mail: metering@stackforce.de