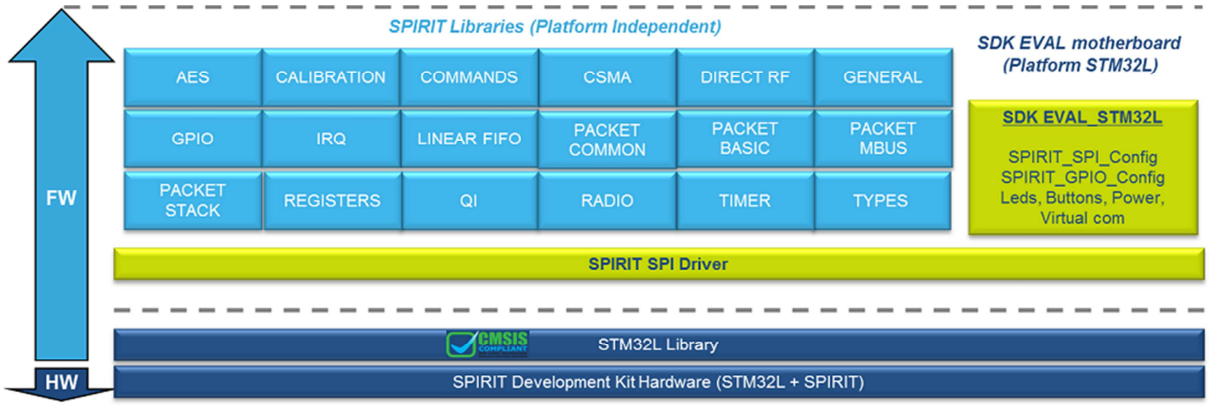
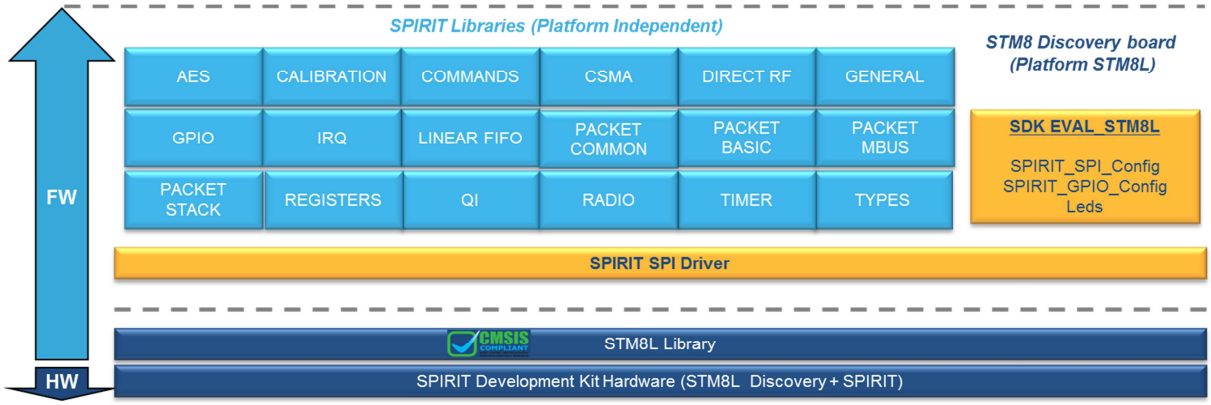
Introduction

This firmware provides a set of APIs to manage the Spirit device using the Spirit Development Kit Eval motherboard and a Virtual Com port driver.

This document provides a description of Spirit Low Level APIs from a general point of view and a more detailed view of the *Spirit1_Libraries.*

To have more details the user should consult the *doxygen* documentation available in the *.chm* file.





These libraries and APIs can be used to build more complex application. The libraries package contains some useful examples to show how to use them.

Naming convention

The following naming convention has been adopted in these modules.

A variable name begins with a lowercase letter. Multiple words are managed as letter-case separate words. A variable name is composed by 3 parts:

<access_specifier><type_prefix>< name>

The *access specifier* identifies the variable scope and is one of:

- “s_” for static variable.
- “g_” for global variable.
- “m_” for class not-static class variable (only for C++ file).

The access specifier is omitted for a local variable.

The *type prefix* encodes the actual data type of the variable. Common used type prefix are:

- ‘l’ for long type.
- ‘n’ for short type.
- ‘f’ for float and double types.
- ‘c’ for char type.
- ‘v’ for void type.
- ‘p’ for pointer type.
- “str” for string type that is a character array.
- “vect” for array type.
- ‘x’ for user defined type.

The *name* is a meaningful multiple words identifier.

Example:

```
SpiritIrqs xIrqStatus;
SRadioInit *pxRadioInit;
static uint32_t s_lXtalFrequency;
uint8_t cRegisterValue;
```

1 SPIRIT1 LIBRARIES

1.1 Introduction

This chapter gives an overview of the Spirit1 library files that can be included when developing an application.

1.2 Spirit1 Libraries Overview

A library is a collection of functions grouped for reference and ease of linking. The Spirit1 Libraries are included in the /inc and /src subdirectories of the folder */Spirit1_Library_Project/Source/SPIRIT1_Libraries.*

There is one module for each macro-feature the Spirit provides and each one is made up of a *.h* and a *.c* files.

See section 1.4 for details.

1.3 Platform Independent Libraries

The Spirit1 Libraries are developed in order to be platform independent. Every function is written in pure C99 code and some of them make use of standard C functions.

Because the SPIRIT1 is driven through an SPI interface, the whole library is based on the idea of refer every routine to a common SPI driver to write or read registers, to send commands and, in general, communicate with the device.

In this way the user can compile these libraries on different platforms as long as he provides the same interface (function prototypes) of the SPI drivers and the implementation of them for the specific platform.

The header file constituting this interface exports their implementations which are called by every *Spirit1_Library* function in order to ensure the portability.

See section 1.22 for more details.

1.4 Modules Overview

The following modules are provided by the *Spirit1 Libraries*:

SPIRIT_Types ***(Section 1.5):*** Contains the definition of data types used in the library and of a global variable which tracks the status of the device updated to the last done transaction.

SPIRIT_Aes ***(Section 1.6):*** provides a set of APIs to manage the 128-bit AES coprocessor available on SPIRIT1.

SPIRIT_Calibration ***(Section 1.7):*** provides a set of APIs to manage the Spirit oscillators calibration and to access to the results provided by the automatic embedded calibration algorithms.

SPIRIT_Commands ***(Section 1.8):*** contains functions and macros used to strobe commands provided by the device logic.

SPIRIT_Csma ***(Section 1.9):*** allows the management of the embedded CSMA algorithm (timers management and max number of retransmissions).

SPIRIT_DirectRf ***(Section 1.10):*** through this module is possible to program the transmission/reception of an arbitrary bitflow regardless of any type of control on them.

SPIRIT_General ***(Section 1.11):*** manages the generic functionalities of Spirit including the reading of the *part name* and *version*.

SPIRIT_Gpio ***(Section 1.12):*** allows the configuration of the Spirit GPIO pins.

SPIRIT_Irq ***(Section 1.13):*** provides a set of APIs for the configuration of those interrupts the device can raise (enabling/disabling and status mask reading).

SPIRIT_LinearFifo ***(Section 1.14):*** manages the TX / RX linear FIFOs.

SPIRIT_Management ***(Section 1.14):*** manages all the workarounds and routines that do not suit to the other modules.

SPIRIT_PktCommon: provides functions to manage the common features of the packets. These functions are overwritten by the *SPIRIT_PktBasic, SPIRIT_PktStack and* modules.

SPIRIT_PktBasic ***(Section 1.16):*** allows the configuration of the Basic packet.

SPIRIT_PktStack ***(Section 1.17):*** allows the configuration of the STack packet.

SPIRIT_PktMbus ***(Section 1.18):*** allows the configuration of the WMBUS packet.

SPIRIT_Qi ***(Section 1.19):*** provides APIs to manage the Quality Indicators the Spirit provides.

SPIRIT_Radio ***(Section 1.20):*** provides a set of APIs to manage the Spirit RF analog and digital parts.

SPIRIT_Timer ***(Section 1.21):*** allows the configuration of the Spirit embedded timers.

Every module includes the file *SPIRIT_Regs.h* containing a map matching every register name with its own address and bitfields.

Moreover two header files are provided:

SPIRIT_Config.h: to be included in the application code and made up of several *#include* directives to be commented by the user in order to include in his code only the interested modules.

MCU_Interface.h ***(Section 1.22)****:* This header file constitutes an interface to the SPI driver used to communicate with Spirit. It exports some function prototypes to write/read registers and FIFOs and to strobe commands. Since the Spirit libraries are totally platform independent, the implementation of these functions are not provided here. The user has to implement these functions taking care to keep the exported prototypes (an example of implementation of Spirit SPI Driver for STM32L is provided in the *SDK_EVAL library*).

1.5 Spirit Types Module

This module contains some useful types definitions in order to make the entire library completely independent from external modules. Each type is completely defined in this module and doesn't use external libraries with the exception of the *stdint.h* and *stdio.h* standard C libraries.

These defined common types are *SpiritFunctionalState (S_ENABLE , S_DISABLE) , SpiritBool (S_TRUE, S_FALSE), SpiritFlagStatus (S_SET, S_RESET)* inspired by the MCD libraries common types.

Moreover, in this module is defined the global variable *g_xStatus* which contains the status of Spirit and is updated every time an SPI transaction occurs. The macro *GET_STATUS()* should be used every time this variable has to be referenced externally.

This variable is of type *SpiritStatus* and represents the single field of the Spirit status returned on each SPI transaction, equal also to the MC_STATE registers. This field-oriented structure allows user to address in simple way the single field of the SPIRIT status.

The fields order in the structure depends on used endianness (little or big endian). The actual definition is valid ONLY for LITTLE ENDIAN mode. Be sure to change opportunely the fields order when use a different endianness.

Moreover the function **SpiritRefreshStatus()** can be called every time the application wants to know the updated *Spirit Status* bytes.

1.6 Aes Module

The user can decide to encrypt data in order to increase the security of his communication system. The AES engine is independent from the transmission/reception procedures, so the user should encrypt his data offline and then do a normal transmission of them.

AES feature is supported by the following functions:

Function	Description
SpiritAesMode	Enables or Disables the AES engine
SpiritAesWriteDataIn	Writes the data to encrypt or decrypt into the AES_DATA_IN registers
SpiritAesReadDataOut	Returns the encrypted or decrypted data from the AES_DATA_OUT register
SpiritAesWriteKey	Writes the encryption key into the AES_KEY_IN register
SpiritAesReadKey	Returns the encryption/decryption key from the AES_KEY_IN register
SpiritAesDeriveDecKeyFromEnc	Derives the decryption key from a given encryption key
SpiritAesExecuteEncryption	Executes the encryption operation
SpiritAesExecuteDecryption	Executes the decryption operation
SpiritAesDeriveDecKeyExecuteDec	Executes the key derivation and the decryption operation

Application examples

In order to encrypt data, the user shall manage the *AES_END* IRQ. The data have to be splitted in blocks of 16 bytes and written into the *AES_DATA_IN* registers. Then, after the key is written into the AES_KEY registers, a command of Execute encryption has to be sent.

Example:

```
#define N_BYTES 12

SpiritFlagStatus aes_end_flag = S_RESET;
uint8_t key_enc[] =
{0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF}
;
uint8_t data_buff[N_BYTES] = {1,2,3,4,5,6,7,8,9,10,11,12} , enc_data_buff[N_BYTES];

SpiritAesMode(S_ENABLE);
SpiritAesWriteKey(key_enc);
SpiritAesWriteDataIn(data_buff , N_BYTES);
SpiritAesExecuteEncryption();

while(!aes_end_flag);       /* the flag is set by the ISR which manages
the AES_END irq */
aes_end_flag=S_RESET;

SpiritAesReadDataOut(enc_data_buff , N_BYTES);
```

In order to decrypt data, the user shall manage the AES_END IRQ and have a decryption key. There are two operative modes to make the data decryption.

Derive the decryption key from the encryption key and decrypt data directly using the *SpiritAesDeriveDecKeyExecuteDec()* function.

Example:

```
#define N_BYTES 12

SpiritFlagStatus aes_end_flag = S_RESET;
uint8_t key_enc[] =
{0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF}
;
uint8_t data_buff[N_BYTES] = {1,2,3,4,5,6,7,8,9,10,11,12};

SpiritAesMode(S_ENABLE);
SpiritAesWriteKey(key_enc);
SpiritAesWriteDataIn(enc_data_buff , N_BYTES);
SpiritAesDeriveDecKeyExecuteDec();

while(!aes_end_flag);       /* the flag is set by the ISR routine which
manages the AES_END irq */
aes_end_flag=S_RESET;

SpiritAesReadDataOut(data_buff,N_BYTES);
```

Derive the decryption key from the encryption key using the *SpiritAesDeriveDecKeyFromEnc()* function, store it into the AES KEY registers and then decrypt data using the *SpiritAesExecuteDecryption()* function.

Example:

```
#define N_BYTES 12

SpiritFlagStatus aes_end_flag = S_RESET;
uint8_t key_enc[] =
{0xA0,0xA1,0xA2,0xA3,0xA4,0xA5,0xA6,0xA7,0xA8,0xA9,0xAA,0xAB,0xAC,0xAD,0xAE,0xAF}
;
uint8_t data_buff[N_BYTES] = {1,2,3,4,5,6,7,8,9,10,11,12};

SpiritAesMode(S_ENABLE);
SpiritAesWriteDataIn(key_enc , 16);

SpiritAesDeriveDecKeyFromEnc();

while(!aes_end_flag);    /* the flag is set by the ISR routine which
manages the AES_END irq */
aes_end_flag=S_RESET;

SpiritAesReadDataOut(key_dec , 16);

SpiritAesWriteKey(key_dec);
SpiritAesWriteDataIn(enc_data_buff , 16);
SpiritAesExecuteDecryption();

while(!aes_end_flag);    /* the flag is set by the ISR routine which
manages the AES_END irq */
aes_end_flag=S_RESET;

/* read the decrypted data */
SpiritAesReadDataOut(data_buff , N_BYTES);
```

1.7 Calibration Module

This module allows the user to set some parameters which deal with the oscillators calibration. The state machine of Spirit contemplates some optional calibrating operations in the transition between the READY and the LOCK state.

RCO/VCO calibration features are supported by the following functions:

Function	Description
SpiritCalibrationRco	Enables or Disables the RCO calibration
SpiritCalibrationVco	Enables or Disables the VCO calibration
SpiritCalibrationSetRcoCalWords	Sets the RCO calibration words
SpiritCalibrationGetRcoCalWords	Returns the RCO calibration words
SpiritCalibrationGetVcoCalData	Returns the VCO calibration data from internal VCO calibrator
SpiritCalibrationSetVcoCalDataTx	Sets the VCO calibration data to be used in TX mode
SpiritCalibrationGetVcoCalDataTx	Returns the actual VCO calibration data used in TX mode
SpiritCalibrationSetVcoCalDataRx	Sets the VCO calibration data to be used in RX mode
SpiritCalibrationGetVcoCalDataRx	Returns the actual VCO calibration data used in RX mode
SpiritCalibrationSetVcoWindow	Sets the VCO calibration window
SpiritCalibrationGetVcoWindow	Returns the VCO calibration window

Application examples

The user can enable or disable the automatic RCO/VCO calibration by calling the functions *SpiritCalibrationVco()* and *SpiritCalibrationRco()*.

The following example shows how to do an initial calibration of VCO, but similar operations can be done also for the RCO calibrator.

Example:

```
uint8_t calData;

SpiritCalibrationVco(S_ENABLE);
SpiritCmdStrobeLockTx();

while(GET_STATUS().MC_STATE != MC_STATE_LOCK){
    SpiritRefreshStatus();
}

calData = SpiritCalibrationGetVcoCalDataTx();
SpiritCalibrationSetVcoCalDataTx(calData);

SpiritCmdStrobeReady();
SpiritCalibrationVco(S_DISABLE);
```

1.8 Commands Module

The user can strobe commands using this module which contains every possible command the Spirit can manage.

Spirit Commands are supported by the following functions and macros:

Function	**Description**
SpiritCmdStrobeCommand	Sends a specific command to SPIRIT
Macro	**Description**
SpiritCmdStrobeTx	Strobes the Tx command
SpiritCmdStrobeRx	Strobes the Rx command
SpiritCmdStrobeReady	Strobes the Ready command
SpiritCmdStrobeStandby	Strobes the Standby command
SpiritCmdStrobeSleep	Strobes the Sleep command
SpiritCmdStrobeLockRx	Strobes the Lock Rx command
SpiritCmdStrobeLockTx	Strobes the Lock Tx command
SpiritCmdStrobeSabort	Strobes the Sabort command
SpiritCmdStrobeLdcReload	Strobes the LDC Reload command
SpiritCmdStrobeSequenceUpdate	Strobes the Sequence update command
SpiritCmdStrobeAesEnc	Strobes the AES encode command
SpiritCmdStrobeAesKey	Strobes the AES key command
SpiritCmdStrobeAesDec	Strobes the AES decode command
SpiritCmdStrobeInitLoad	Strobes the INIT LOAD command
SpiritCmdStrobeSres	Strobes the reset command
SpiritCmdStrobeFlushRxFifo	Strobes the Flush Rx command
SpiritCmdStrobeFlushTxFifo	Strobes the Flush Tx command

Application examples

Every command strobe is an SPI transaction with a specific command code.

Example:

```
…

SpiritCmdStrobeRx();    /* Send the command RX to Spirit */

…
```

1.9 CSMA Module

In CSMA (Carrier Sense Multiple Access) mode a node verifies the absence of other traffic before transmitting. The Spirit CSMA feature, when configured and enabled, is transparent for the user.

The module is provided with the structure *CsmaInit* which contains all the parameters used to configure the CSMA mechanism.

CSMA feature is supported by the following functions:

Function	Description
SpiritCsmaInit	Initializes the Spirit CSMA
SpiritCsmaGetInfo	Returns the fitted structure *CsmaInit* starting from the registers values
SpiritCsma	Enables/Disables the CSMA mode
SpiritGetCsma	Gets the CSMA mode. Says if it is enabled or disabled
SpiritCsmaPersistentMode	Enables/Disables the CSMA persistent mode
SpiritCsmaGetPersistentMode	Gets the persistent CSMA mode
SpiritCsmaSeedReloadMode	Enables/Disables the CSMA seed reload mode
SpiritCsmaGetSeedReloadMode	Gets the seed reload mode
SpiritCsmaSetBuCounterSeed	Writes the BU counter seed
SpiritCsmaGetBuCounterSeed	Returns the BU counter seed
SpiritCsmaSetBuPrescaler	Writes the BU prescaler
SpiritCsmaGetBuPrescaler	Returns the BU prescaler
SpiritCsmaSetCcaPeriod	Writes the CCA period
SpiritCsmaGetCcaPeriod	Returns the CCA period
SpiritCsmaSetCcaLength	Writes the CCA length
SpiritCsmaGetCcaLength	Returns the CCA period
SpiritCsmaSetMaxNumberBackoff	Writes the max number of back-off
SpiritCsmaGetMaxNumberBackoff	Returns the max number of back-off

Application examples

The user has only to call the *SpiritCsmaInit()* function on a filled structure and enable the CSMA policy using the *SpiritCsma()* function.

Example:

```
CsmaInit csmaInit={
       S_DISABLE,        /* not persistent mode */
       TBIT_TIME_64,     /* Tbit time */
       TCCA_TIME_3,      /* Tcca time */
       5,                /* max number of backoffs */
       0xFA21,           /* BU counter seed */
       32                /* BU prescaler */
};
```

```
SpiritCsmaInit(&csmaInit);

...

// enable CSMA before the transmission starts
SpiritCsma(S_ENABLE);

SpiritCmdStrobeTx();

...

// and disable after the transmission has been completed
SpiritCsma(S_DISABLE);

...
```

1.10 Direct RF Module

This module contains functions to manage the direct Tx/Rx mode. This mode can be used to transmit a bit flow without any additional field and skipping the controls of the packet handler.

Direct RF feature is supported by the following functions:

Function	Description
SpiritDirectRfSetRxMode	Sets the DirectRF RX mode of SPIRIT
SpiritDirectRfGetRxMode	Returns the DirectRF RX mode of SPIRIT
SpiritDirectRfSetTxMode	Sets the TX source of SPIRIT
SpiritDirectRfGetTxMode	Returns the DirectRF TX mode of SPIRIT

Application examples

Data can be provided by FIFO or through a configured GPIO in which data have to be clocked. The user can choose the way to send data to Spirit through the enumerative types *DirectTx* and *DirectRx*.

Example:

```
…

SpiritDirectRfSetTxMode(DIRECT_TX_FIFO_MODE);

…
```

1.11 General Module

Spirit General features are supported by the following functions:

Function	Description
SpiritGeneralBatteryLevel	Enables/Disables the output of battery level detector
SpiritGeneralSetBatteryLevel	Sets the battery level
SpiritGeneralGetBatteryLevel	Returns the battery level.
SpiritGeneralBrownOut	Enables/Disables the output of brown out detector
SpiritGeneralHighPwr	Enables/Disables the high power mode
SpiritGeneralSetExtRef	Sets External Reference
SpiritGeneralGetExtRef	Returns External Reference
SpiritGeneralSetXoGm	Sets XO gm
SpiritGeneralGetXoGm	Returns XO gm
SpiritGeneralGetPktType	Returns the settled packet format
SpiritGeneralGetDevicePartNumber	Returns the Spirit part number
SpiritGeneralSetSpiritVersion	Sets the Spirit RF board version number
SpiritGeneralGetSpiritVersion	Returns the Spirit RF board version number
SpiritGeneralLibraryVersion	Returns the current library version as a string

Application examples

This module provides functions for some Spirit generic features.

The example shows take the spirit version number. This number is not really coded in a register of the device but it is stored into a static variable (inside this module) and can be one of the values of the *SpiritVersion* enumerative *typedef*.

The user can use it to develope applications with different behaviors in relation to this number.

Example:

```
uint8_t versionNumber;

…

/* get the device version number */
versionNumber = SpiritGeneralGetSpiritVersion();

…
```

1.12 Gpio Module

This module can be used to configure the Spirit Gpio pins to perform specific functions. The structure *SGpioInit* can be used to specify these features for one of the four Spirit Gpio pin.

Spirit GPIO management is supported by the following functions:

Function	Description
SpiritGpioInit	Initializes the SPIRIT GPIOx
SpiritGpioTemperatureSensor	Enables or Disables the output of temperature sensor
SpiritGpioSetLevel	Forces to VDD or GND a SPIRIT GPIOx configured as digital output
SpiritGpioGetLevel	Returns the output value (VDD or GND) of SPIRIT GPIOx
SpiritGpioClockOutputInit	Initializes the SPIRIT Clock Output
SpiritGpioClockOutput	Enables or disables the clock on pin output
SpiritGpioSetXOPrescaler	Sets the XO ratio as clock output
SpiritGpioGetXOPrescaler	Returns the settled XO prescaler as clock output
SpiritGpioSetRCOPrescaler	Sets the RCO ratio as clock output
SpiritGpioGetRCOPrescaler	Returns the RCO ratio as clock output
SpiritGpioSetExtraClockCycles	Sets the extra clock cycles
SpiritGpioGetExtraClockCycles	Returns the extra clock cycles

Application examples

The following example shows how to configure the Spirit GPIO_3 as an IRQ source for a microcontroller using the SpiritGpioInit() function.

Example:

```
...

SGpioInit gpioIRQ={
  SPIRIT_GPIO_3,   /*  configure the Spirit GPIO_3 */
  SPIRIT_GPIO_MODE_DIGITAL_OUTPUT_LP, /* set low power output */
  SPIRIT_GPIO_DIG_OUT_IRQ  /* configure the output in  IRQ mode */
};

...

/* configure the Spirit registers according to the SGpioInit
structure */
SpiritGpioInit(&gpioIRQ);

...
```

1.13 Irq Module

On the Spirit side, specific IRQs can be enabled by setting a specific bitmask. Setting to 1 a specific bit, the corresponding IRQ will be notified through a voltage signal on a configured GPIO (see section 1.12).

The module is provided with two typedef

- *SpiritIrqs*: a bitfield structure the user can fit in order to enable specific IRQs
- *IrqList*: a typedef enumeration with all the possible IRQs the Spirit can raise

Spirit IRQs management is supported by the following functions:

Function	Description
`SpiritIrqDeInit`	De-initializes the SpiritIrqs structure setting all the bitfield to 0
`SpiritIrqInit`	Enables all the IRQs according to the user defined *SpiritIrqs* structure
`SpiritIrq`	Enables or disable a specific IRQ using the an element of the *IrqList* typedef
`SpiritIrqGetMask`	Fills a pointer to a structure of *SpiritIrqs* type reading the IRQ_MASK register
`SpiritIrqGetStatus`	Fills a pointer to a structure of *SpiritIrqs* type reading the IRQ_STATUS registers
`SpiritIrqClearStatus`	Clears the IRQ status registers
`SpiritIrqCheckFlag`	Verifies if a specific IRQ has been generated

Application examples

The Spirit libraries allow the user to do this in two different ways:

The first enables the IRQs one by one, i.e. using an SPI transaction for each IRQ to enable.

Example:

```
...

SpiritIrqDeInit(NULL);   /* this call is used to reset the IRQ
mask register */

/* enable some IRQs */
SpiritIrq(RX_DATA_READY , S_ENABLE);
SpiritIrq(VALID_SYNC , S_ENABLE);
SpiritIrq(RX_TIMEOUT , S_ENABLE);

...
```

The second strategy is to set the IRQ bitfields structure. So, during the initialization the user has to fill the *SpiritIrqs* structure setting to one the single field related to the IRQ he wants to enable, and to zero (default) the single field related to all the IRQs he wants to disable.

Example:

```
...

SpiritIrqs irqMask={0};

 ...

/* fill the SpiritIrqs structure */
irqMask.IRQ_RX_DATA_READY = 1;
irqMask.IRQ_VALID_SYNC = 1;
irqMask.IRQ_RX_TIMEOUT = 1;

/* set the registers according to the SpiritIrqs structure */
SpiritIrqDeInit(&irqMask);

...
```

The most applications will require a Spirit IRQ notification on a microcontroller EXTI line. Then, the user can check which irq raised using two different ways into the ISR of the EXTI line physically linked to the Spirit pin configured for IRQ:

- Check **only one** Spirit IRQ (because the Spirit IRQ status register automatically blanks itself after an SPI reading) into the ISR.

Example:

```
...

if(SpiritIrqCheckFlag(RX_DATA_READY))
 {
          /* do something... */
 }

...
```

- Check more than one Spirit IRQ status by storing the entire IRQ status registers into a bitfields *SpiritIrqs* structure and then check the interested bits.

Example:

```
...

SpiritIrqGetStatus(&irqStatus);

 if(irqStatus.IRQ_RX_DATA_READY)
 {
        /* do something... */
 }
 if(irqStatus.IRQ_VALID_SYNC)
 {
        /* do something... */
 }
 if(irqStatus.RX_TIMEOUT)
 {
```

```
        /* do something... */
 }

...
```

1.14 Linear Fifo Module

This module allows the user to manage the linear FIFO. The APIs exported here can be used to set the thresholds for the FIFO almost full / empty alarm interrupts or to get the total number of elements inside the FIFO.

Spirit linear FIFO management is supported by the following functions:

Function	Description
SpiritLinearFifoReadNumElementsRxFifo	Returns the number of elements in the RX FIFO
SpiritLinearFifoReadNumElementsTxFifo	Returns the number of elements in the TX FIFO
SpiritLinearFifoSetAlmostFullThresholdRx	Sets the almost full threshold for the RX FIFO
SpiritLinearFifoGetAlmostFullThresholdRx	Returns the almost full threshold for RX FIFO
SpiritLinearFifoSetAlmostEmptyThresholdRx	Sets the almost empty threshold for the RX FIFO
SpiritLinearFifoGetAlmostEmptyThresholdRx	Returns the almost empty threshold for RX FIFO
SpiritLinearFifoSetAlmostFullThresholdTx	Sets the almost full threshold for the TX FIFO
SpiritLinearFifoGetAlmostFullThresholdTx	Returns the almost full threshold for TX FIFO
SpiritLinearFifoSetAlmostEmptyThresholdTx	Sets the almost empty threshold for the TX FIFO
SpiritLinearFifoGetAlmostEmptyThresholdTx	Returns the almost empty threshold for TX FIFO

Application examples

The almost full thresholds are encountered from the top of the FIFO while the almost empty thresholds are encountered from the start side. The following example shows how to use the set threshold functions.

Example:

```
...

SpiritLinearFifoSetAlmostEmptyThresholdRx(5); /* sets the AE
threshold for the Rx fifo to 5 elements */

SpiritLinearFifoSetAlmostFullThresholdTx(9);   /* sets the AF
threshold for the Tx fifo to 96-9 = 87 elements */

...
```

1.15 Management Module

This module exports some functions used to manage all the things that are considered optimizations of the way of working of the device.

Here follows a table of the exported functions.

Function	Description
SpiritManagementWaVcoCalibration	VCO calibration routine. It is recommended to be used in order to find the optimal value of the calibration word for the selected frequency. Called by the radio init function.
SpiritManagementWaCmdStrobeTx	Recommended when a TX command is sent.
SpiritManagementWaCmdStrobeRx	Recommended when a RX command is sent. Sets the CWC bank capacitor optimally according to the band.
SpiritManagementWaTRxFcMem	Used to store an internal state in this file. To be called before *SpiritManagementWaVcoCalibration* to set correctly the desired frequency.
SpiritManagementWaExtraCurrent	To be called at the SHUTDOWN exit. It avoids extra current consumption at SLEEP and STANDBY.

1.16 Basic Packet Module

This module overrides some functions from the *SPIRIT_PktCommon* module. These APIs are exported as macros to redefine their name in according to the used naming convention.

The module can be used to manage the configuration of Spirit Basic packets. The user can obtain a packet configuration filling the structure *PktBasicInit*, defining in it some general parameters for the Spirit Basic packet format.

Another structure the user can fill is *PktBasicAddressesInit* to define the addresses which will be used during the communication.

Spirit packet Basic management is supported by the following functions and macros:

Function	**Description**
`SpiritPktBasicInit`	Initializes the SPIRIT Basic packet
`SpiritPktBasicGetInfo`	Returns the SPIRIT Basic packet structure according to the specified parameters in the registers
`SpiritPktBasicAddressesInit`	Initializes the SPIRIT Basic packet addresses
`SpiritPktBasicGetAddressesInfo`	Returns the SPIRIT Basic packet addresses structure according to the specified parameters in the registers
`SpiritPktBasicSetFormat`	Configures the Basic packet format as packet used automatically by SPIRIT
`SpiritPktBasicAddressField`	Enables or disables the ADDRESS field for SPIRIT Basic packets
`SpiritPktBasicGetAddressField`	Notifies if the ADDRESS field is enabled or disabled
`SpiritPktBasicSetPayloadLength`	Sets payload length for SPIRIT Basic packets
`SpiritPktBasicGetPayloadLength`	Returns payload length for SPIRIT Basic packets
`SpiritPktBasicSetVarLengthWidth`	Computes and sets the variable payload length for SPIRIT Basic packets
Macro	**Description**
`SpiritPktBasicSetControlLength`	Sets the CONTROL field length for SPIRIT Basic packets
`SpiritPktBasicGetControlLength`	Returns the CONTROL field length for SPIRIT Basic packets
`SpiritPktBasicSetPreambleLength`	Sets the PREAMBLE field Length mode for SPIRIT Basic packets
`SpiritPktBasicGetPreambleLength`	Returns the PREAMBLE field Length mode for SPIRIT Basic packets.
`SpiritPktBasicSetSyncLength`	Sets the SYNC field Length for SPIRIT Basic packets
`SpiritPktBasicGetSyncLength`	Returns the SYNC field Length for

	SPIRIT Basic packets
SpiritPktBasicSetFixVarLength	Sets fixed or variable payload length mode for SPIRIT Basic packet
SpiritPktBasicFilterOnCrc	If set the RX packet is discarded when CRC is not valid
SpiritPktBasicGetFilterOnCrc	Returns the CRC filtering bit
SpiritPktBasicSetCrcMode	Sets the CRC poly to use for SPIRIT Basic packets
SpiritPktBasicGetCrcMode	Returns the CRC poly used for SPIRIT Basic packets
SpiritPktBasicWhitening	Enables/Disables WHITENING for SPIRIT packets
SpiritPktBasicFec	Enables/Disables FEC for SPIRIT packets
SpiritPktBasicSetSyncxWord	Sets a specific SYNC word (1 byte) for SPIRIT Basic packets
SpiritPktBasicGetSyncxWord	Returns a specific SYNC words used for SPIRIT Basic packets
SpiritPktBasicSetSyncWords	Sets all the SYNC words for SPIRIT Basic packets
SpiritPktBasicGetSyncWords	Returns all the SYNC words for SPIRIT Basic packets
SpiritPktBasicGetVarLengthWidth	Returns the SPIRIT variable length width
SpiritPktBasicSetDestinationAddress	Sets the destination address for the Tx packet
SpiritPktBasicGetTransmittedDestAddress	Returns the RX packet source address
SpiritPktBasicSetMyAddress	Writes TX packet source address
SpiritPktBasicGetMyAddress	Returns TX packet source address
SpiritPktBasicSetBroadcastAddress	Writes BROADCAST packet source address
SpiritPktBasicGetBroadcastAddress	Returns BROADCAST packet source address
SpiritPktBasicSetMulticastAddress	Writes MULTICAST packet source address
SpiritPktBasicGetMulticastAddress	Returns MULTICAST packet source address
SpiritPktBasicSetCtrlMask	Sets the control mask
SpiritPktBasicGetCtrlMask	Returns the control mask
SpiritPktBasicSetCtrlReference	Sets the control field reference
SpiritPktBasicGetCtrlReference	Returns the control field reference
SpiritPktBasicSetTransmittedCtrlField	Sets the TX control field
SpiritPktBasicGetTransmittedCtrlField	Returns the TX control field
SpiritPktBasicFilterOnMyAddress	Enables/Disables the filtering on My Address
SpiritPktBasicFilterOnMulticastAddress	Enables/Disables the filtering on

	Multicast Address
SpiritPktBasicFilterOnBroadcastAddress	Enables/Disables the filtering on Broadcast Address
SpiritPktBasicGetFilterOnMyAddress	Returns the filtering on My Address bit
SpiritPktBasicGetFilterOnMulticastAddress	Returns the filtering on Multicast Address bit
SpiritPktBasicGetFilterOnBroadcastAddress	Returns the filtering on Broadcast Address bit
SpiritPktBasicGetReceivedDestAddress	Returns the destination address of the received packet
SpiritPktBasicGetReceivedCtrlField	Returns the control field of the received packet
SpiritPktBasicGetReceivedCrcField	Returns the CRC field of the received packet
SpiritPktBasicGetReceivedPktLength	Returns the packet length of the received packet
SpiritPktBasicFilterOnControlField	Enables or disables the filtering on CONTROL fields
SpiritPktBasicGetFilterOnControlField	Returns the filtering on CONTROL fields enable bit

Application examples

The following example basically shows how to configure a Basic packet and its address. Moreover it configures also the payload length and the destination address.

Example:

```
...

PktBasicInit basicInit={
    PKT_PREAMBLE_LENGTH_08BYTES,  /* preamble length in bytes */
    PKT_SYNC_LENGTH_4BYTES,       /* sync word length in bytes */
    0x1A2635A8,           /* sync word */
    PKT_LENGTH_VAR,       /* variable or fixed payload length */
    7,  /* length field width in bits (used only for variable
length) */
    PKT_NO_CRC,           /* CRC mode */
    PKT_CONTROL_LENGTH_0BYTES, /* control field length */
    S_ENABLE,            /* address field */
    S_DISABLE,           /* FEC */
    S_ENABLE             /* whitening */
  };

PktBasicAddressesInit addressInit={
    S_ENABLE,  /* enable/disable filtering on my address */
    0x34,      /* my address (address of the current node) */
    S_DISABLE, /* enable/disable filtering on multicast address */
    0xEE,      /* multicast address */
    S_DISABLE, /* enable/disable filtering on broadcast address */
```

```
    0xFF       /* broadcast address */
  };

 ...

 /* set the spirit packet registers using the filled structure */
 SpiritPktBasicInit(&basicInit);
 /* set the spirit address registers using the filled structure */
 SpiritPktBasicAddressesInit(&addressInit);

 ...

/* set the spirit packet length */
SpiritPktBasicSetPayloadLength(20);

/* set the spirit destination address */
SpiritPktBasicSetDestinationAddress(0x44);

 ...
```

1.17 STack Packet Module

This module overrides some functions from the *SPIRIT_PktCommon* module. These APIs are exported as macros to redefine their name in according to the used naming convention.

This module can be used to manage the configuration of Spirit STack packets, and it is quite similar to the Basic packets one since the STack packets can be considered an extension of the Basic one. The user can obtain a packet configuration filling the structure *PktStackInit*, defining in it some general parameters for the Spirit STack packet format. Another structure the user shall fill is *PktStackAddressesInit* to define the addresses which will be used during the communication. The structure *PktStackLlpInit* is provided in order to configure the link layer protocol features like autoack, autoretransmission or piggybacking.

Spirit packet STack management is supported by the following functions and macros:

Function	Description
`SpiritPktStackInit`	Initializes the SPIRIT STack packet
`SpiritPktStackGetInfo`	Returns the SPIRIT STack packet structure according to the specified parameters in the registers
`SpiritPktStackAddressesInit`	Initializes the SPIRIT STack packet addresses
`SpiritPktStackAddressesGetInfo`	Returns the SPIRIT STack packet addresses structure according to the specified parameters in the registers
`SpiritPktStackLlpInit`	Initializes the SPIRIT STack packet LLP options
`SpiritPktStackLlpGetInfo`	Returns the SPIRIT STack packet LLP structure according to the specified parameters in the registers
`SpiritPktStackSetFormat`	Configures the STack packet format for SPIRIT
`SpiritPktStackAddressField`	Enables or disables the presence of Address field for SPIRIT STack packets
`SpiritPktStackGetAddressField`	Notifies if the Address field is enabled or disabled
`SpiritPktStackSetPayloadLength`	Sets payload length for SPIRIT STack packets
`SpiritPktStackGetPayloadLength`	Returns payload length for SPIRIT STack packets
`SpiritPktStackSetVarLengthWidth`	Computes and sets the variable payload length for SPIRIT STack packets
`SpiritPktStackSetRxSourceMask`	Sets the Rx packet source mask
`SpiritPktStackGetRxSourceMask`	Returns the Rx packet source mask

SpiritPktStackGetReceivedPktLength	Returns the packet length field of the received packet
SpiritPktStackFilterOnSourceAddress	Enables/Disables the filtering on Source Address
Macro	**Description**
SpiritPktStackSetControlLength	Sets the CONTROL field length for SPIRIT STack packets
SpiritPktStackGetControlLength	Returns the CONTROL field length for SPIRIT STack packets
SpiritPktStackSetPreambleLength	Sets the PREAMBLE field length mode for SPIRIT STack packets
SpiritPktStackGetPreambleLength	Returns the PREAMBLE field length mode for SPIRIT STack packets
SpiritPktStackSetSyncLength	Sets the SYNC field length for SPIRIT STack packets
SpiritPktStackGetSyncLength	Returns the SYNC Length for SPIRIT STack packets
SpiritPktStackSetFixVarLength	Sets fixed or variable payload length mode for SPIRIT STack packets
SpiritPktStackFilterOnCrc	Enables/Disables the filtering on CRC
SpiritPktStackGetFilterOnCrc	Returns the CRC filtering bit
SpiritPktStackSetCrcMode	Sets the CRC mode for SPIRIT STack packets
SpiritPktStackGetCrcMode	Returns the CRCmode for SPIRIT STack packets
SpiritPktStackWhitening	Enables/Disables WHITENING for SPIRIT STack packets
SpiritPktStackFec	Enables/Disables FEC for SPIRIT STack packets
SpiritPktStackSetSyncxWord	Sets a specific SYNCx word for SPIRIT STack packets
SpiritPktStackGetSyncxWord	Returns a specific SYNCx word for SPIRIT STack packets
SpiritPktStackSetSyncWords	Sets all the SYNC words for SPIRIT STack packets
SpiritPktStackGetSyncWords	Returns all the SYNC words for SPIRIT STack packets
SpiritPktStackGetVarLengthWidth	Returns the SPIRIT variable length width
SpiritPktStackSetDestinationAddress	Sets the destination address for the Tx packet
SpiritPktStackSetSourceReferenceAddress	Sets the RX packet reference source address
SpiritPktStackGetSourceReferenceAddress	Returns the RX packet reference source address
SpiritPktStackGetTransmittedDestAddress	Returns the RX packet source

	address
SpiritPktStackSetMyAddress	Writes TX packet source address
SpiritPktStackGetMyAddress	Returns TX packet source address
SpiritPktStackSetBroadcastAddress	Writes BROADCAST packet source address
SpiritPktStackGetBroadcastAddress	Returns BROADCAST packet source address
SpiritPktStackSetMulticastAddress	Writes the MULTICAST packet source address
SpiritPktStackGetMulticastAddress	Returns the MULTICAST packet source address
SpiritPktStackSetCtrlMask	Sets the control mask
SpiritPktStackGetCtrlMask	Returns the control mask
SpiritPktStackSetCtrlReference	Sets the control field reference
SpiritPktStackGetCtrlReference	Returns the control field reference
SpiritPktStackSetTransmittedCtrlField	Sets the TX control field
SpiritPktStackGetTransmittedCtrlField	Returns the TX control field
SpiritPktStackFilterOnMyAddress	Enables/Disables the filtering on My Address
SpiritPktStackFilterOnMulticastAddress	Enables/Disables the filtering on Multicast Address
SpiritPktStackFilterOnBroadcastAddress	Enables/Disables the filtering on Broadcast Address
SpiritPktStackGetFilterOnMyAddress	Returns the filtering on My Address bit
SpiritPktStackGetFilterOnMulticastAddress	Returns the filtering on Multicast Address bit
SpiritPktStackGetFilterOnBroadcastAddress	Returns the filtering on Broadcast Address bit
SpiritPktStackGetReceivedCtrlField	Returns the control field of the received packet
SpiritPktStackGetReceivedCrcField	Returns the CRC field of the received packet
SpiritPktStackGetReceivedPktLength	Returns the packet length of the received packet
SpiritPktStackFilterOnControlField	Enables or disables the filtering on CONTROL fields
SpiritPktStackGetFilterOnControlField	Returns the filtering on CONTROL fields enable bit
SpiritPktStackAutoAck	Sets the AUTO ACKNOWLEDGMENT mechanism on the receiver
SpiritPktStackRequireAck	Sets the AUTO ACKNOWLEDGMENT mechanism on the transmitter

SpiritPktStackSetTransmittedSeqNumberReload	Sets the TX sequence number to be used to start counting
SpiritPktStackSetNMaxReTx	Sets the max number of max retransmissions
SpiritPktStackGetNMaxReTx	Returns the max number of max retransmissions
SpiritPktStackGetReceivedDestAddress	Returns the destination address of the received packet
SpiritPktStackGetReceivedSourceAddress	Returns the source address of the received packet
SpiritPktStackGetReceivedSeqNumber	Returns the sequence number of the received packet
SpiritPktStackGetReceivedNackRx	Returns the Nack bit of the received packet
SpiritPktStackGetTransmittedSeqNumber	Returns the sequence number of the transmitted packet
SpiritPktStackGetNReTx	Returns the number of retransmission done on the transmitted packet

Application examples

The following example basically shows how to configure a Basic packet, its address and eventual LLP features.

Moreover it configures also the payload length and the destination address.

Example:

```
PktStackInit stackInit={
   PKT_PREAMBLE_LENGTH_08BYTES,  /* preamble length in bytes */
   PKT_SYNC_LENGTH_4BYTES,   /* sync word length in bytes */
   0x1A2635A8,       /* sync word */
   PKT_LENGTH_VAR,   /* variable or fixed payload length
   7,     /* length field width in bits (used only for variable length)*/
   PKT_NO_CRC,       /* CRC mode */
   PKT_CONTROL_LENGTH_0BYTES, /* control field length */
   S_DISABLE,        /* FEC */
   S_ENABLE          /* whitening */
 };

PktStackAddressesInit addressInit={
   S_ENABLE,         /* enable/disable filtering on my address */
   0x34,             /* my address (address of the current node) */
   S_DISABLE,        /* enable/disable filtering on multicast address */
   0xEE,             /* multicast address */
   S_DISABLE,        /* enable/disable filtering on broadcast address */
   0xFF              /* broadcast address */
};

PktStackLlpInit stackLLPInit ={
  S_DISABLE,         /* enable/disable the autoack feature */
  S_DISABLE,         /* enable/disable the piggybacking feature */
```

```
  PKT_DISABLE_RETX   /* set the max number of retransmissions or disable
them */
};

 /* set the spirit packet registers using the filled structure */
 SpiritPktStackInit(&stackInit);

 /* set the spirit address registers using the filled structure */
 SpiritPktStackAddressesInit(&addressInit);


 /* set the spirit LLP registers using the filled structure */
 SpiritPktStackLlpInit(&stackLLPInit);

...

/* set the spirit packet length */
SpiritPktStackSetPayloadLength(20);

/* set the spirit destination address */
SpiritPktStackSetDestinationAddress(0x44);

...
```

1.18 MBUS Packet Module

This module can be used to manage the configuration of Spirit MBUS packets. The user can obtain a packet configuration filling the structure *PktMbusInit*, defining in it some general parameters for the Spirit MBUS packet format.

Spirit packet MBUS management is supported by the following functions and macros:

Function	Description
SpiritPktMbusInit	Initializes the SPIRIT MBUS packet
SpiritPktMbusGetInfo	Returns the SPIRIT MBUS packet structure according to the specified parameters in the registers
SpiritPktMbusSetFormat	Configures the MBUS packet format as the one used by SPIRIT
SpiritPktMbusSetPreamble	Configures the preamble
SpiritPktMbusGetPreamble	Returns the added chip sequence “01” in the preamble

SpiritPktMbusSetPostamble	Configures the postamble
SpiritPktMbusGetPostamble	Returns the added chip sequence “01” in the postamble
SpiritPktMbusSetSubmode	Sets on of the possible sub-mode of MBUS
SpiritPktMbusGetSubmode	Returns the settled MBUS sub-mode
SpiritPktMbusSetPayloadLength	Sets payload length for SPIRIT MBUS packets
SpiritPktMbusGetPayloadLength	Returns payload length for SPIRIT MBUS packets

Application examples

Since the MBUS protocol is a standard, the low level configuration of a MBUS packet is very simple to do.

Example:

```
...

PktMbusInit mbusInit={
  MBUS_SUBMODE_S1_S2_LONG_HEADER,    /* Mbus submode selection */
  36,                                /* preamble additive "01" chips */
  16                                 /* postamble length in "01" chips */
};

 ...

 /* set the spirit packet registers using the filled structure */
 SpiritPktMbusInit(&mbusInit);

 ...
```

1.19 QI Module

This module can be used to configure and read some quality indicators used by Spirit. Spirit QI management is supported by the following functions and macros:

Function	Description
SpiritQiPqiCheck	Enables/Disables the Preamble Quality Indicator (PQI) check
SpiritQiSqiCheck	Enables/Disables the Synchronization Quality Indicator (SQI) check
SpiritQiSetPqiThreshold	Sets the PQI threshold
SpiritQiGetPqiThreshold	Returns the PQI threshold
SpiritQiSetSqiThreshold	Sets the SQI threshold

SpiritQiGetSqiThreshold	Returns the SQI threshold
SpiritQiSetRssiThreshold	Sets the RSSI threshold
SpiritQiGetRssiThreshold	Returns the RSSI threshold
SpiritQiComputeRssiThreshold	Computes the RSSI threshold from its dBm value
SpiritQiSetRssiThresholddBm	Sets the RSSI threshold from its dBm value
SpiritQiGetPqi	Returns the PQI value
SpiritQiGetSqi	Returns the SQI value
SpiritQiGetCs	Returns the CS state
SpiritQiGetRssi	Returns the RSSI value
SpiritQiSetRssiFilterGain	Sets the RSSI filter gain
SpiritQiGetRssiFilterGain	Returns the RSSI filter gain
SpiritQiSetCsMode	Sets the CS Mode
SpiritQiGetCsMode	Returns the CS Mode
SpiritQiCsTimeoutMask	Enables/Disables the CS Timeout Mask
SpiritQiPqiTimeoutMask	Enables/Disables the PQI Timeout Mask
SpiritQiSqiTimeoutMask	Enables/Disables the SQI Timeout Mask
Macro	**Description**
SpiritQiGetRssidBm	Macro to obtain the RSSI value in dBm

Application examples

APIs to set thresholds and to read values in raw mode or in dBm are provided.

Example:

```
float rssiValueDbm;
uint8_t pqiValue, sqiValue;

/* enable the SQI and PQI check */
SpiritQiPqiCheck(S_ENABLE);
SpiritQiSqiCheck(S_ENABLE);

...

/* RSSI, PQI and SQI readings */
rssiValueDbm = SpiritQiGetRssidBm();
pqiValue = SpiritQiGetPqi();
sqiValue = SpiritQiGetSqi();

...
```

1.20 Radio Module

This module should be used for the configuration and management of SPIRIT RF Analog and Digital part.

An important data structure is exported by this module and is used to configure all the fundamental radio parameters.

Spirit Radio management is supported by the following functions:

Function	Description
SpiritRadioInit	Initializes the SPIRIT analog and digital radio part
SpiritRadioGetInfo	Returns the SPIRIT analog and digital radio structure according to the registers value
SpiritRadioSetXtalFlag	Sets the Xtal configuration
SpiritRadioGetXtalFlag	Returns the Xtal configuration
SpiritRadioSearchWCP	Returns the charge pump word
SpiritRadioSetSynthWord	Sets the SYNTH registers
SpiritRadioGetSynthWord	Returns the SYNTH word
SpiritRadioSetBand	Sets the operating band
SpiritRadioGetBand	Returns the operating band
SpiritRadioSetChannel	Sets the channel number
SpiritRadioGetChannel	Returns the channel number
SpiritRadioSetChannelSpace	Sets the channel space
SpiritRadioGetChannelSpace	Returns the channel space
SpiritRadioSetFrequencyOffsetPpm	Sets the FC OFFSET register starting from a value in ppm
SpiritRadioSetFrequencyOffset	Sets the FC OFFSET register starting from a value in Hz
SpiritRadioGetFrequencyOffset	Returns the actual frequency offset in Hz
SpiritRadioSetFrequencyBase	Sets the Synth word and the Band Select register according to desired base carrier frequency
SpiritRadioGetFrequencyBase	Returns the base carrier frequency
SpiritRadioGetCenterFrequency	Returns the actual channel center frequency
SpiritRadioSearchDatarateME	Returns the datarate mantissa and exponent
SpiritRadioSearchFreqDevME	Returns the frequency deviation mantissa and exponent
SpiritRadioSearchChannelBwME	Returns the channel bandwidth mantissa and exponent
SpiritRadioSetDatarate	Sets the datarate
SpiritRadioGetDatarate	Returns the datarate
SpiritRadioSetFrequencyDev	Sets the frequency deviation
SpiritRadioGetFrequencyDev	Returns the frequency deviation

SpiritRadioSetChannelBW	Sets the channel bandwidth
SpiritRadioGetChannelBW	Returns the channel bandwidth
SpiritRadioSetModulation	Sets the modulation scheme
SpiritRadioGetModulation	Returns the modulation scheme
SpiritRadioCWTransmitMode	Enables or Disables the Continuous Wave transmit mode
SpiritRadioSetOokPeakDecay	Sets the OOK Peak Decay
SpiritRadioGetOokPeakDecay	Returns the OOK Peak Decay
SpiritRadioSetPATabledBm	Configures the Power Amplifier Table starting from values in dBm
SpiritRadioGetPATabledBm	Returns the Power Amplifier Table returning values in dBm
SpiritRadioSetPATable	Configures the Power Amplifier Table and registers
SpiritRadioGetPATable	Returns the Power Amplifier Table and registers
SpiritRadioSetPALeveldBm	Sets a specific PA_LEVEL register with a value given in dBm
SpiritRadioGetPALeveldBm	Returns a specific PA_LEVEL register in dBm
SpiritRadioSetPALevel	Sets a specific PA_LEVEL register
SpiritRadioGetPALevel	Returns a specific PA_LEVEL register
SpiritRadioSetPACwc	Sets the output stage additional load capacitor bank
SpiritRadioGetPACwc	Returns the output stage additional load capacitor bank
SpiritRadioSetPALevelMaxIndex	Sets a specific PA_LEVEL_MAX_INDEX
SpiritRadioGetPALevelMaxIndex	Returns the PA_LEVEL_MAX_INDEX
SpiritRadioSetPAStepWidth	Sets a specific PA_RAMP_STEP_WIDTH
SpiritRadioGetPAStepWidth	Returns the PA_RAMP_STEP_WIDTH
SpiritRadioPARamping	Enables or Disables the Power Ramping
SpiritRadioGetPARamping	Returns the Power Ramping enable bit
SpiritRadioAFC	Enables or Disables the AFC
SpiritRadioAFCFreezeOnSync	Enables or Disables the AFC freeze on sync word detection
SpiritRadioSetAFCMode	Sets the AFC working mode
SpiritRadioGetAFCMode	Returns the AFC working mode
SpiritRadioSetAFCPDLeakage	Sets the AFC peak detector leakage
SpiritRadioGetAFCPDLeakage	Returns the AFC peak detector leakage
SpiritRadioSetAFCFastPeriod	Sets the length of the AFC fast period
SpiritRadioGetAFCFastPeriod	Returns the length of the AFC fast period
SpiritRadioSetAFCFastGain	Sets the AFC loop gain in fast mode

SpiritRadioGetAFCFastGain	Returns the AFC loop gain in fast mode
SpiritRadioSetAFCSlowGain	Sets the AFC loop gain in slow mode
SpiritRadioGetAFCSlowGain	Returns the AFC loop gain in slow mode
SpiritRadioGetAFCCorrectionReg	Returns the AFC correction from the corresponding register
SpiritRadioGetAFCCorrectionHz	Returns the AFC correction expressed in Hz
SpiritRadioAutoSetFOffset	Corrects the frequency offset through the AFC mechanism
SpiritRadioAGC	Enables or Disables the AGC
SpiritRadioSetAGCMeasureTimeUs	Sets the AGC measure time in us
SpiritRadioGetAGCMeasureTimeUs	Returns the AGC measure time in us
SpiritRadioSetAGCMeasureTime	Sets the AGC measure time
SpiritRadioGetAGCMeasureTime	Returns the AGC measure time
SpiritRadioSetAGCHighThreshold	Sets the AGC high threshold
SpiritRadioGetAGCHighThreshold	Returns the AGC high threshold
SpiritRadioSetAGCLowThreshold	Sets the AGC low threshold
SpiritRadioGetAGCLowThreshold	Returns the AGC low threshold
SpiritRadioSetClkRecMode	Sets the clock recovery algorithm
SpiritRadioGetClkRecMode	Returns the Clock Recovery working mode
SpiritRadioSetClkRecPGain	Sets the clock recovery proportional gain
SpiritRadioGetClkRecPGain	Returns the clock recovery proportional gain
SpiritRadioSetClkRecIGain	Sets the clock recovery integral gain
SpiritRadioGetClkRecIGain	Returns the clock recovery integral gain
SpiritRadioSetClkRecPstFltLength	Sets the postfilter length
SpiritRadioGetClkRecPstFltLength	Returns the postfilter length
SpiritRadioCsBlanking	Enables or Disables the received data blanking when the CS is under the threshold
SpiritRadioPersistenRx	Turns the radio in persistent Rx mode
SpiritRadioGetXtalFrequency	Gets the Xtal frequency
SpiritRadioSetXtalFrequency	Sets the Xtal frequency [1]
SpiritRadioSetRefDiv	Sets the reference divider
SpiritRadioGetRefDiv	Gets the reference divider
SpiritRadioSetDigDiv	Sets the digital divider
SpiritRadioGetDigDiv	Gets the digital divider

Application examples

In order to configure the Radio main parameters, the user shall fit the *SRadioInit* structure and call the *SpiritRadioInit()* function passing its pointer as argument.

Before that the XTAL frequency must be mandatory set, otherwise the radio won't be configured properly.

[1] This function must be called necessary before the radio configuration.

Example:

```
...

SRadioInit radioInit = {
     0,                  /* Xtal offset in ppm */
     433.4e6,            /* base frequency */
     20e3,               /* channel space */
     0,                  /* channel number */
     FSK,                /* modulation select */
     38400,              /* datarate */
     20e3,               /* frequency deviation */
     100.5e3             /* channel filter bandwidth */
};

...

/* suppose to have a 50MHz XTAL */
SpiritRadioSetXtalFrequency(50000000)


/* set the spirit radio registers using the filled structure */
SpiritRadioInit(&radioInit);

...
```

Another important parameter for the radio configuration is the transmission power. The user is allowed to configure it using the function *SpiritRadioSetPALeveldBm()* which sets the PA LEVEL specified by the first argument to the power expressed in dBm by the second parameter.

Example:

```
...

SpiritRadioSetPALeveldBm(0 , -10.0);    /* set power to -10 dBm
*/
SpiritRadioSetPALevelMaxIndex(0);       /* say to the radio
module to take the Pa level 0 as output power */

...
```

The effective power that is set can be a little different from the passed argument in dBm because the function performs a linear approximation given by a specific fitting formula.

1.21 Timer Module

This module provides APIs to configure the Spirit timing mechanisms. They allow the user to set the timer registers using raw values or passing the desired timer value expressed in ms.

Moreover the management of the Spirit LDCR mode can be done using these APIs.

Spirit timers management is supported by the following functions and macros:

Function	**Description**
SpiritTimerLdcrMode	Enables/Disables the LDCR mode
SpiritTimerLdcrAutoReload	Enables or Disables the LDCR timer reloading
SpiritTimerLdcrGetAutoReload	Returns the LDCR timer reload bit
SpiritTimerSetRxTimeout	Sets the RX timeout timer initialization registers
SpiritTimerSetRxTimeoutMs	Sets the RX timeout timer counter and prescaler from the desired value in ms
SpiritTimerSetRxTimeoutCounter	Sets the RX timeout timer counter
SpiritTimerSetRxTimeoutPrescaler	Sets the RX timeout timer prescaler
SpiritTimerGetRxTimeout	Returns the RX timeout
SpiritTimerSetWakeUpTimer	Sets the Wake Up timeout timer initialization registers
SpiritTimerSetWakeUpTimerMs	Sets the Wake Up timeout timer counter and prescaler from the desired value in ms
SpiritTimerSetWakeUpTimerCounter	Sets the Wake Up timeout timer counter
SpiritTimerSetWakeUpTimerPrescaler	Sets the Wake Up timeout timer prescaler
SpiritTimerGetWakeUpTimer	Returns the Wake Up timeout
SpiritTimerSetWakeUpTimerReloadMs	Sets the Wake Up timer reload initialization registers
SpiritTimerSetWakeUpTimerReload	Sets the Wake Up timer reload counter and prescaler from the desired value in ms
SpiritTimerSetWakeUpTimerReloadCounter	Sets the Wake Up timer reload counter
SpiritTimerSetWakeUpTimerReloadPrescaler	Sets the Wake Up timer reload prescaler
SpiritTimerGetWakeUpTimerReload	Returns the Wake Up timer reload value
SpiritTimerComputeWakeUpValues	Computes the values of the wakeup timer
SpiritTimerComputeRxTimeoutValues	Computes the values of the RX timer
SpiritTimerSetRxTimeoutStopCondition	Sets the Rx timeout stop conditions
SpiritTimerReloadStrobe	Strobes the Reload command
SpiritTimerGetRcoFrequency	Returns the RCO frequency in Hz (assuming that it has been calibrated)
Macro	**Description**

SET_INFINITE_RX_TIMEOUT	Sets an infinite Rx timeout

Application examples

The following example shows how to use the APIs to configure the LDCR mode.

Example:

```
...
/* configure the Rx and the wakeup timeout */
SpiritTimerSetRxTimeoutMs(50.0);
SpiritTimerSetWakeUpTimerMs(150.0);

/* here the user should put the IRQ configuration for
RX_TIMEOUT and WAKEUP_TIMEOUT */

...

/* start the LDCR mode */
SpiritTimerLdcrMode(S_ENABLE);

...
```

1.22 MCU interface Module

The Spirit Library provides the following functions prototypes to manage the SPI communication with a microcontroller.

Function	Description
SpiritSpiInit	Initializes the MCU SPI to drive SPIRIT1
SpiritSpiWriteRegisters	Write single or multiple SPIRIT register
SpiritSpiReadRegisters	Read single or multiple SPIRIT register
SpiritSpiCommandStrobes	Send a command
SpiritSpiWriteLinearFifo	Write data into TX FIFO
SpiritSpiReadLinearFifo	Read data from RX FIFO
SpiritEnterShutdown	Turn Spirit1 OFF
SpiritExitShutdown	Turn Spirit1 ON
SpiritCheckShutdown	Check the Spirit1 status

Important note: These prototypes are placed into the MCU_Interface.h file, but the implementation of these functions are placed in the *SDK_EVAL_Spi_Driver.c* file and are written for the used STM32L microcontroller (mounted in the SDK eval motherboard). The user who wants to make the porting of the Spirit libraries on another platform has to implement these functions with respect of signature (name, arguments and return type) for the used microcontroller.

1.23 Overview on a complete base configuration

After the initialization of the microcontroller, the user should configure the physical and data link layers of Spirit1. This can be done using the Spirit1 Libraries. The mandatory steps to obtain a correct configuration will follow here. In these code examples, all the parameters (of data structures or function arguments) are passed through some defined symbols that are supposed to be assigned before. See the specific sections of this manual or the Spirit1 Library examples for further details.

- **Radio configuration:**

  Basically, two functions must be called. The first is the XTAL frequency setting. It is needed in order to set a library internal variable that will be used to correctly compute all the frequency and time parameters. If not set, all the further settings done by the Spirit1 Library won't work properly.

  After that, the SpiritRadioInit function, will configure the radio physical layer. All the parameters of the SpiritRadioInit structure must be chosen within the physical limits (datarate in relation to the frequency deviation and Rx channel filter), otherwise the radio will not work properly.

```
SRadioInit xRadioInit = {
   XTAL_OFFSET_PPM,
   BASE_FREQUENCY,
   CHANNEL_SPACE,
   CHANNEL_NUMBER,
   MODULATION_SELECT,
   DATARATE,
   FREQ_DEVIATION,
   BANDWIDTH
 };

SpiritRadioSetXtalFrequency(XTAL_FREQUENCY);
SpiritRadioInit(&xRadioInit);
```

  For the transmitter configuration the power level must be set. In case of frequency or OOK modulations, only one level of the PA table is important to be set (ex. the 0 level). This level must be specified with the call SpiritRadioSetPALevelMaxIndex.

```
SpiritRadioSetPALeveldBm(0,POWER_DBM);
SpiritRadioSetPALevelMaxIndex(0);
```

  In case of ASK modulation, all the PA levels must be set and the power ramping must be enabled by the call to SpiritRadioPARamping.

- **Packet configuration**

To configure the packet parameters a format must be chosen (Basic as in the following example, STack, MBUS or a direct mode in order to skip the packet handler control).

```
PktBasicInit xBasicInit={
  PREAMBLE_LENGTH,
  SYNC_LENGTH,
  SYNC_WORD,
  LENGTH_TYPE,
  LENGTH_WIDTH,
  CRC_MODE,
  CONTROL_LENGTH,
  EN_ADDRESS,
  EN_FEC,
  EN_WHITENING
};


PktBasicAddressesInit xAddressInit={
  EN_FILT_MY_ADDRESS,
  MY_ADDRESS,
  EN_FILT_MULTICAST_ADDRESS,
  MULTICAST_ADDRESS,
  EN_FILT_BROADCAST_ADDRESS,
  BROADCAST_ADDRESS
};

SpiritPktBasicInit(&xBasicInit);
SpiritPktBasicAddressesInit(&xAddressInit);
```

- **IRQ configuration**

On the Spirit1 side, the IRQ configuration is done in two steps.
The first is the Spirit-GPIO configuration that configures a specific pin (among the four GPIOs of the device) to generate a falling edge when a specific interrupt arises.
An example can be the enable of the RX DATA READY that reports a packet correctly received. After the IRQ line configuration and before starting any transaction, the reading of the IRQ status registers is always recommended.

```
SGpioInit xGpioIRQ={
  SPIRIT_GPIO_3,
  SPIRIT_GPIO_MODE_DIGITAL_OUTPUT_LP,
  SPIRIT_GPIO_DIG_OUT_IRQ
};


SpiritGpioInit(&xGpioIRQ);
SpiritIrqDeInit(NULL);
```

```
SpiritIrq(RX_DATA_READY);
SpiritIrqClearStatus();
```

- **Timer configuration**

  On the receiver side, the user can configure the Rx timeout and the criterion that must be used for it to be stopped (ex. SQI threshold). These operations can be done using the following instructions.

```
SpiritTimerSetRxTimeoutMs(1000.0);
SpiritTimerSetRxTimeoutStopCondition(SQI_ABOVE_THRESHOLD);
```

  An infinite Rx timeout can be chosen using the macro SET_INFINITE_RX_TIMEOUT **.**

- **Transmission**

  To perform a transmission the user should:

  - Load the Tx FIFO with the data to be sent
  - Set the payload length
  - Strobe a Tx command
  - Wait for the TX_DONE IRQ (that should have been previously configured and currently managed by the microcontrolled).

```
SpiritSpiWriteLinearFifo(N, tx_buff);
SpiritPktBasicSetPayloadLength(N);
SpiritCmdStrobeTx();

/* wait for the Tx Done IRQ. Here it just sets the
tx_data_sent_flag to 1 */
while(!tx_data_sent_flag);
tx_data_sent_flag=0;
```

- **Reception**

  In order to receive the user should:

  - Set the expected payload length (only in case of fixed length packet)
  - Strobe a Rx command
  - Wait for the RX_DONE IRQ (that should have been previously configured and currently managed by the microcontrolled).
  - Read the Rx FIFO length obtaining the number (N) of received bytes.
  - Read N bytes from the Rx FIFO.

```
SpiritPktBasicSetPayloadLength(N); /* only for fixed len */
SpiritCmdStrobeRx();

/* wait for the Tx Done IRQ. Here it just sets the
rx_data_received_flag to 1 */
while(!rx_data_received_flag);
rx_data_received_flag=0;

N=SpiritLinearFifoReadNumElementsRxFifo();
SpiritSpiReadLinearFifo(N, rx_buff);
```

- **Other settings**

On the receiver side, the Spirit1 allows the user to perform filters based on addresses constraints (like destination address and also source in case of the STack format), CRC, Control general purpose fields.

On the transmitter CSMA can be configured and keep enabled when the device reaches the TX state.

See the specific sections of this manual or the library attached examples for further details.

2 SDK EVAL LIBRARIES

2.1 Introduction

This chapter gives an overview of the *SDK Eval Library* files that can be included in an application.

2.2 SDK Eval Library Overview

The SDK EVAL library is the low level driver that allows controlling the SPIRIT1 by our demo-kit motherboards in which the STM32L microcontroller is used. The SDK EVAL Library are included in the /inc and /src subdirectories of the folder /STM32L/SDK_Eval_STM32L/.

There is one module for each board peripheral and every one of them is made up of .h and .c files.

An implementation for STM8L also is provided and can be used as example of porting to a different microcontroller. The SPIRIT1 library example application can be run on STM8L using a STM8L DISCOVERY board just using a couple of wires to connect a STEVAL-IKR002Vx or STEVAL-IKR001Vx RF modules.

See following sections for details.

2.3 STM32L driver library

Since the demo-kit motherboard mounts a STM32L151RB, the *STM32L library* is included in all the presented modules.

This module supports both the motherboard and the dongle evaluation board.

2.3.1 Modules Overview

Here is documented the hardware peripheral functions found in the *SdkEval library*:

- *SDK_EVAL_Button:* Provides function to configure and manage the SDK Eval board buttons.
- *SDK_EVAL_Led:* Leds management and configuration.
- *SDK_EVAL_PM:* Power management module used to regulate the supply voltage of Spirit.
- *SDK_EVAL_ Gpio:* Microcontroller GPIO configuration.
- *SDK_EVAL_Spi_Driver:* Implementation of the *MCU_Interface.h* exported APIs for the SDK Eval board.
- *SDK_EVAL_Timers:* Management of the STM32L timers.
- *SDK_EVAL_Com:* Management of the UART COM port.

2.3.2 Sdk Eval Button Module

This module exports functions used to configure and manage the SDK motherboard push-buttons.

Function	Description

SdkEvalPushButtonInit	Configures Button GPIO and EXTI Line
SdkEvalPushButtonGetState	Returns the selected Button state

2.3.3 Application examples

This example shows how to configure the microcontroller GPIO associated to the BUTTON_SCM_PS to work in interrupt mode.

Example:

```
...

SdkEvalPushButtonInit(BUTTON_SCM_PS, BUTTON_MODE_EXTI);

...
```

2.3.4 Sdk Eval Led Module

In this module there are APIs for the management of the leds on the SDK Eval motherboard.

Function	Description
SdkEvalLedInit	Configures led GPIOs
SdkEvalLedOn	Turns on the selected led
SdkEvalLedOff	Turns off the selected led
SdkEvalLedToggle	Toggles the selected led
SdkEvalLedGetState	Return the status of a specified led

2.3.5 Application examples

This example shows how to call functions to toggle the SDK board LED1.

Example:

```
 SdkEvalLedInit(LED1);

...

 SdkEvalLedToggle(LED1);

...
```

2.3.6 Sdk Eval PM Module (only STEVAL-IKR001Vx RF motherboard)

This module can be used to regulate the supply voltage of the SPIRIT1; this function can be used only in the STEVAL-IKR001Vx RF motherboard in which a programmable resistor is used. Reading the actual voltage through the MCU ADC it is possible to change it through the digital potentiometer.

This module also provides APIs to read the current supplied to the Spirit.

Function	Description
SdkEvalPmI2CInit	Configures the I2C interface and the correspondent GPIO pins
SdkEvalPmDigipotWrite	Writes an 8-bit value in the RDAC register
SdkEvalPmDigipotRead	Returns the RDAC register
SdkEvalPmADCInit	Configures the Analog-to-Digital Converter and the correspondent GPIO pins
SdkEvalPmGetV	Samples, converts and returns the voltage sample on the specified ADC channel
SdkEvalPmGetSettledV	Samples, converts and returns the mean voltage in a time period on the specified ADC channel
SdkEvalPmRegulateVoltage	Implements a control loop to make the specified voltage equal to a reference one using an incremental regulator
SdkEvalPmRegulateVoltageI	Implements a control loop to make the specified voltage equal to a reference one using an integral regulator (recommended)
SdkEvalPmRfSwitchInit	Configures the RF supply voltage switch GPIO
SdkEvalPmRfSwitchToVRf	Sets the switch to supply RF voltage
SdkEvalPmRfSwitchToRcal	Sets the switch to supply voltage for the calibration resistor
Macro	**Description**
SdkEvalPmGetVRf	Gets the instant RF supply voltage in Volt
SdkEvalPmGetV5V	Gets the instant measured 5V voltage in Volt
SdkEvalPmGetV3V	Gets the instant measured 3V voltage in Volt
SdkEvalPmGetIR1	Gets the first current instant measurement (gain = 31 mV/mA)
SdkEvalPmGetIR2	Gets the second current instant measurement (gain = 50 mV/uA)
SdkEvalPmGetSettledVRf	Gets the averaged RF supply voltage in Volt
SdkEvalPmGetSettledV5V	Gets the measured and averaged 5V voltage in Volt
SdkEvalPmGetSettledV3V	Gets the measured and averaged 3V voltage in Volt
SdkEvalPmGetSettledIR1	Gets the first averaged current measurement (gain = 31 mV/mA)
SdkEvalPmGetSettledIR2	Gets the second averaged current measurement (gain = 50 mV/uA)
SdkEvalPmRegulateVRf	Regulates the V RF voltage using the incremental control loop
SdkEvalPmRegulateVRfI	Regulates the V RF voltage using the integral control loop (recommended)

2.3.7 Application examples

The following example shows how to call functions in order to control the Spirit supply voltage and set it to 2.5V.

Example:

```
...

   SdkEvalPmADCInit();
   SdkEvalPmI2CInit();

...

   SdkEvalPmRegulateVRfI(2.5);

...
```

NOTE: The functions *SdkEvalPmRfSwitchInit()* and *SdkEvalPmRfSwitchToVRf()* are always to be called at the beginning of every user application.

2.3.8 Sdk Eval Gpio Module

This module exports API to manage the Spirit GPIO from the micro side. It's also provided two functions to put the Spirit in shutdown or to turn it on.

Function	Description
SdkEvalM2SGpioInit	Configures MCU GPIOs and EXTI Line connected to SPIRIT GPIOs
SdkEvalM2SGpioInterruptCmd	Enables or disables the interrupt on GPIO
SdkEvalSpiritGpioGetLevel	Returns the level of a specified GPIO
SdkEvalSpiritEnterShutdown	Puts in shutdown mode SPIRIT
SdkEvalSpiritExitShutdown	Forces out SPIRIT from shutdown mode

2.3.9 Application examples

The following example shows how to configure the shutdown pin and the GPIO 3 as an EXTI input.

Example:

```
...

 SdkEvalM2SGpioInit(M2S_GPIO_SDN,M2S_MODE_GPIO_OUT);

 SdkEvalM2SGpioInit(M2S_GPIO_3,M2S_MODE_EXTI_IN);
 SdkEvalM2SGpioInterruptCmd(M2S_GPIO_3,0x0A,0x0A,ENABLE);

 ...

 SdkEvalSpiritExitShutdown();

 ...
```

2.3.10 Sdk Eval Timers

This module allows the user to easily configure the STM32L timers. The APIs that are provided are limited to the generation of an IRQ every time the timer elapses.

Function	Description
SdkEvalTimersFindFactors	Computes two integer value prescaler and period such that *Cycles = prescaler * period*
SdkEvalTimersTimConfig	Configures the specified timer to raise an interrupt every time the counter reaches specified period value counting with a specified prescaler
SdkDelayMs	Implements a delay using the microcontroller Systick with a step of 1 ms
SdkDelay10Us	Implements a delay using the microcontroller Systick with a step of 10 us
Macro	**Description**
SdkEvalTimersTimConfig_ms	Configures the specified TIMER to raise an interrupt every TIME ms.
SdkEvalTimersState	ENABLES or DISABLES a specified timer
SdkEvalTimersSetCounter	Sets a specific counter for the specified timer
SdkEvalTimersResetCounter	Resets the counter of the specified timer

2.3.11 Application examples

This example shows how to configure the microcontroller *TIMER2* to raise an interrupt every 60 ms.

Example:

```
...

SdkEvalTimersTimConfig_ms(TIM2,60.0);

...

SdkEvalTimersState(TIM2, ENABLE);          /* the timer
starts counting here */

...

SdkEvalTimersState(TIM2, DISABLE);         /* timer
stopped */

...
```

2.4 STM8L driver library

The SPIRIT1 library is a platform independent firmware driver and it is easy to make a porting to another microcontroller. This operation requires to re-write the low level driver (SPI and GPIO functions mainly) for another microcontroller. In all the demo-kit motherboards the STM32L is used and the firmware package included in the release supports the STM32L in the folder STM32L/SDK_Eval_STM32L/, but also the STM8L is supported in the folder STM8L/SDK_Eval_STM8L/.

In SDK_Eval_STM8L library the main functions to drive the SPIRIT1 are defined, in particular the modules:

- *SDK_EVAL_Led:* Leds management and configuration.
- *SDK_EVAL_ Gpio:* Microcontroller GPIO configuration.
- *SDK_EVAL_Spi_Driver:* Implementation of the MCU_Interface.h exported APIs for the SDK Eval board.
- *SDK_EVAL_Timers:* Management of the STM8L timers.
- *SDK_EVAL_Com:* Management of the UART COM port.

2.4.1 Sdk Eval Led Module

In this module there are APIs for the management of the leds on the SDK Eval motherboard.

Function	Description
`SdkEvalLedInit`	Configures led GPIOs
`SdkEvalLedOn`	Turns on the selected led
`SdkEvalLedOff`	Turns off the selected led
`SdkEvalLedToggle`	Toggles the selected led
`SdkEvalLedGetState`	Return the status of a specified led

2.4.2 Sdk Eval Gpio Module

This module exports API to manage the Spirit GPIO from the micro side. It's also provided two functions to put the Spirit in shutdown or to turn it on.

Function	Description
`SdkEvalM2SGpioInit`	Configures MCU GPIOs and EXTI Line connected to SPIRIT GPIOs
`SdkEvalM2SGpioInterruptCmd`	Enables or disables the interrupt on GPIO
`SdkEvalSpiritGpioGetLevel`	Returns the level of a specified GPIO
`SdkEvalSpiritEnterShutdown`	Puts in shutdown mode SPIRIT
`SdkEvalSpiritExitShutdown`	Forces out SPIRIT from shutdown mode

2.4.3 Sdk Eval Timers

This module allows the user to easily configure the STM32L timers. The APIs that are provided are limited to the generation of an IRQ every time the timer elapses.

Function	Description
SdkEvalTimersFindFactors	Computes two integer value prescaler and period such that *Cycles = prescaler * period*
SdkEvalTimersTimConfig	Configures the specified timer to raise an interrupt every time the counter reaches specified period value counting with a specified prescaler
SdkDelayMs	Implements a delay using the microcontroller Systick with a step of 1 ms
SdkDelay10Us	Implements a delay using the microcontroller Systick with a step of 10 us
Macro	**Description**
SdkEvalTimersTimConfig_ms	Configures the specified TIMER to raise an interrupt every TIME ms.
SdkEvalTimersState	ENABLES or DISABLES a specified timer
SdkEvalTimersSetCounter	Sets a specific counter for the specified timer
SdkEvalTimersResetCounter	Resets the counter of the specified timer

2.4.4 Example hardware

The STM8L driver library is a basic implementation for STM8L and this library has been used with the evaluation board STM8L-DISCOVERY board, based on the STM8L152C6T6.

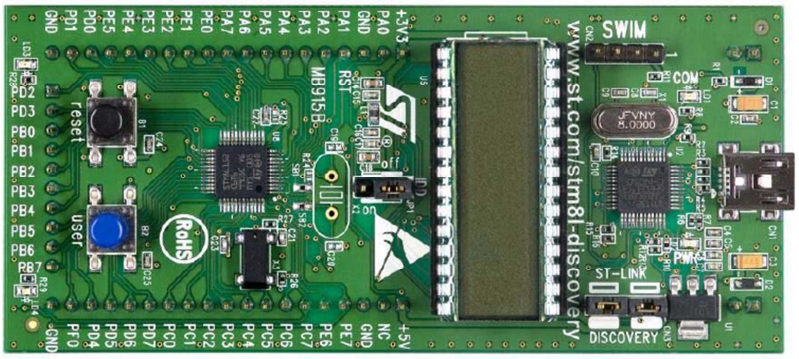


Figure 1

The SPIRIT1 daughter board (STEVAL-IKR001Vx and STEVAL-IKR002Vx) needs the connection with the SPI port of the microcontroller and 5 GPIOs: 1 for the SDN pin of the SPIRIT1 and 4 for the GPIOs of the SPIRIT1 (not mandatory).

Here below the pin connection adopted in the SDK Eval STM8L library:

Signal name	STEVAL-IKR002Vx RF module	STM8L-DISCOVERY CONNECTOR	STM8L pin name
SPI CSn	J7, pin 3	P3, pin 7	PB4
SPI SCLK	J7, pin 5	P3, pin 8	PB5
SPI MOSI	J7, pin 9	P3, pin 9	PB6
SPI MISO	J7, pin 7	P3, pin 10	PB7

SPIRIT1 SDN	J6, pin 9	P2, pin 12	PC1
SPIRIT1 GPIO_0	J7, pin 10	P2, pin 4	PE7
SPIRIT1 GPIO_1	J7, pin 8	P2, pin 5	PE6
SPIRIT1 GPIO_2	J7, pin 6	P2, pin 6	PC7
SPIRIT1 GPIO_3	J7, pin 4	P2, pin 9	PC4
SPIRIT1 GND	J6, pin 5/7	GND	GND
SPIRIT1 VBAT	J6, pin 8/10	3.3 VDD	3.3 VDD

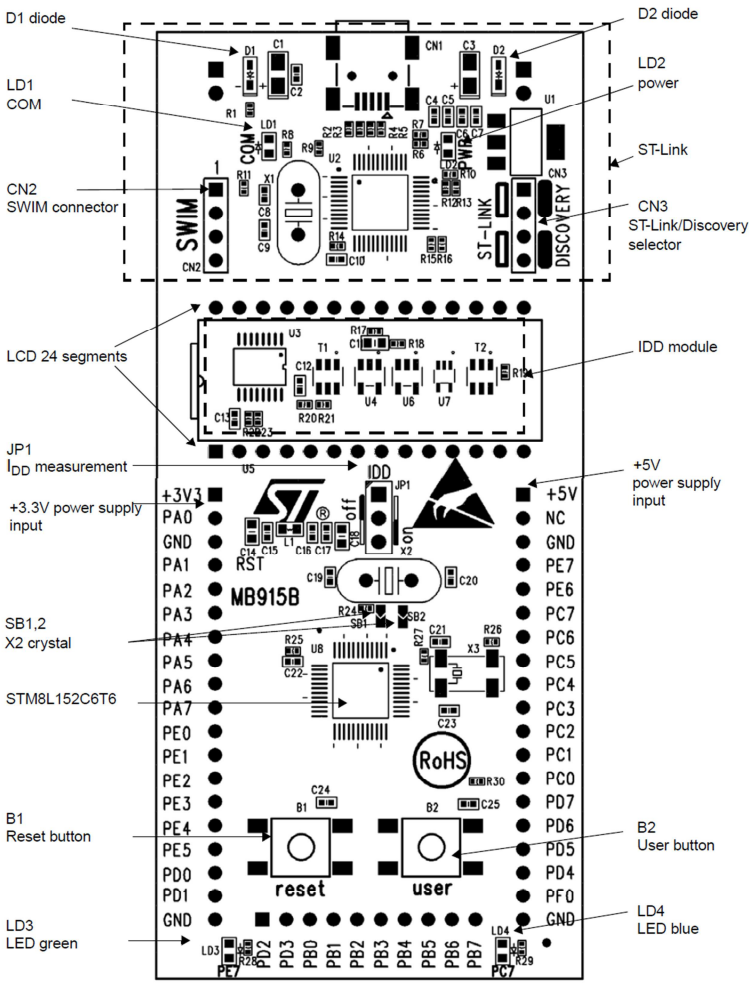


Figure 2

3 IAR PROJECT OVERVIEW

3.1 Introduction

This chapter gives an overview of the IAR project provided with the *Spirit Low Level* libraries. The *IAR Systems EWARM v.7.20 (*www.iar.com) has been used in order to make easier the building and flashing of the application examples provided with this packet.

3.2 Iar Project Ovewview

The IAR project included in the folder /Spirit1_Library_Project/Source/SPIRIT1_Libraries/IAR can be opened by the file *Spirit_Library_Project.eww*.

The project has all the necessary links to the library folders required to make the application examples work and, moreover, there's an specific configuration for each example.. This allows the user to test the different examples simply by selecting the corresponding configuration, building and flashing the micro.

3.3 Examples Overview

The examples source code together with readme files can be found in the directory */SPIRIT1_Library_Project/Application/examples* and consist in:

- *Basic generic*: simple transmission and reception of Basic packets with a fixed payload.
- *CSMA*: example using Spirit CSMA .
- *Encryption*: example of communication of encrypted data (using the Spirit AES engine).
- *FIFO Handler*: Transmission and reception of packets that can have a payload longer than 96 byte.
- *LDCR*: Example of low duty cycle communication mode implemented using the embedded mechanism.
- *Ping Pong*: communication between two devices, each one switches between Tx and Rx.
- *Sniff*: Spirit1 in sniff mode. It is a very rubust low power consumption strategy that allows the reception of packets without synchronization with the Tx.
- *STack generic*: transmission and reception of STack packets with a fixed payload. Used to test the filtering features of the device.
- *STack LLP*: Packets transmission/reception example using the LLP features of Stack (ACK/ Autoretransmission).
- *WMBUS standard*: transmission and reception of MBUS link layer standard A-packets with a fixed payload.

TABLE OF CONTENTS