TWN4

API Reference

DocRev16, July 7, 2017

Elatec GmbH

Contents

1 System Functions

1.1 SysCall

This function is useful for writing interfaces, which do a remote call of a system function,

```
bool SysCall(TEnvSysCall *Env);
```

Parameters:	
`TEnvSysCall *Env`	Pointer to a structure which specifies parameters of the functions to be called.
Return:	If the function has been called the return value is `true`, otherwise it is `false`. In this case the specified function does not exist.

1.2 Reset

This functions is performing a reset of the firmware, which also includes a restart of the currently running App.

```
void Reset(void);
```

Parameters:	None.
Return:	None.

1.3 StartBootloader

This function is performing a manual call of the boot loader. As a consequence the execution of the App is stopped.

```
void StartBootloader(void);
```

Parameters:	None.
Return:	None.

1.4 GetSysTicks

Retrieve number of system ticks, specified in multiple of 1 milliseconds, since startup of the firmware.

```
unsigned long GetSysTicks(void);
```

Parameters:	None.
Return:	Number of system ticks since startup of the firmware. The returned value will restart at 0 after 2^{32} system ticks (around 1193 hours).

1.5 GetVersionString

Retrieve version information. The function generates a ASCII string, terminated by 0.

```
int GetVersionString(char *VersionString,int MaxLen);
```

Parameters:	
`char *VersionString`	Pointer to an array of characters, which will receive the version information.
`int MaxLen`	Maximum number of characters, the specified byte array can receive excluding the 0-termination.
Return:	Length of the returned string excluding the 0-termination.

Example:

```
// This sample demonstrates, how to send the version string
// to the host
void WriteChar(char Char)
{
    HostWriteByte(Char);
}
void WriteString(const char *String)
{
    while (*String)
        WriteChar(*String++);
}
void WriteVersion(void)
{
    char Version[30+1];
    GetVersionString(Version,sizeof(Version)-1);
    WriteString(Version);
}
```

1.6 GetUSBType

Retrieve type of USB communication. This could by keyboard emulation or CDC emulation or some other value for future or custom implementations.

```
int GetUSBType(void);
```

Parameters:	None.
Return:	USBTYPE_NONE: No USB stack, USBTYPE_CDC: CDC device (virtual COM port), USBTYPE_KEYBOARD: HID keyboard, USBTYPE_CCID_HID: CCID + HID (compound device), USBTYPE_REPORTS: CCID + HID reports, USBTYPE_CCID_CDC: CCID + CDC (compound device), USBTYPE_CCID: CCID

1.7 GetDeviceType

Retrieve type of underlying TWN4 hardware.

```
int GetDeviceType(void);
```

Parameters:	None.
Return:	DEVTYPE_LEGICNFC: TWN4 LEGIC, DEVTYPE_MIFARENFC: TWN4 MIFARE

1.8 Sleep

The device enters the sleep state for a specified time. During sleep state, the device reduces the current consumption to a value, which depends on the mode of sleep.

```
int Sleep(unsigned long Ticks,unsigned long Flags)
```

Parameters:	
unsigned long Ticks	Time, specified in milliseconds, the device should enter the sleep state.
unsigned long Flags	Events, which cause the function immediately to return. The parameter is a bitwise OR of all events to be handled.
Return:	This function always return the value 0.

Definition	Value	Description
SLEEPMODE_SLEEP	0x0000	During sleep, device still can be waked up via communication port. In this mode, the device has higher current consumption: TWN4 MIFARE: 8mA TWN4 LEGIC: 20mA
SLEEPMODE_STOP	0x0100	During stop, device still cannot be waked up. This results in lower current consumption: TWN4 MIFARE: 0.5mA TWN4 LEGIC: 13.5mA

The sleep mode can optionally be interrupted by events. The events are bitwise or-combined and are specified as parameters in the call of the function Sleep. Following events are defined:

Definition	Value	Description
WAKEUP_BY_USB_MSK	0x01	The USB input channel received at least on byte.
WAKEUP_BY_COM1_MSK	0x02	The input channel of COM1 received at least on byte.
WAKEUP_BY_COM2_MSK	0x04	The input channel of COM2 received at least on byte.
WAKEUP_BY_TIMEOUT_MSK	0x10	Sleep time ran out.
WAKEUP_BY_LPCD_MSK	0x20	The presence of a transponder card was detected. (Supported by TWN4 MultiTech Nano only)

1.9 GetDeviceUID

This function returns a UID, which is unique to the specific TWN4 device.

```
void GetDeviceUID(byte *UID)
```

Parameters:	
byte *UID	Pointer ro an array of bytes, which receives 12 bytes. These 12 bytes represent the UID of the device.
Return:	None.

1.10 SetParameters

This function allows to set parameters, which influence the behaviour of the TWN4 firmware. See also chapter System Parameters for a description of the TLV list and all available paramaters.

```
bool SetParameters(const byte *TLV,int ByteCount)
```

Parameters:	
const byte *TLV	Pointer to an array of bytes, which contains the TLV list.
int ByteCount	Length counted in bytes, the TLV list contains.
Return:	The function returns `true`, if the parameters was set to the new value. Otherwise the function returns `false`.

Example:

```
// This sample demonstrates a call of function SetParameters.
const byte TLVBytes[] =
{
    ICLASS_READMODE, 1, ICLASS_READMODE_PAC,    // Read PAC from iClass.
    INDITAG_READMODE, 1, INDITAG_READMODE_2,    // Set Inditag readmode 2
    TLV_END                                     // End of TLV
};

int main(void)
{
    // ...
    SetParameters(TLVBytes,sizeof(TLVBytes));
    // ...
}
```

1.11 GetLastError

This function allows to read the last error code, which was generated by any system function. For a list of available error code see chapter System Errors.

```
unsigned int GetLastError(void)
```

Parameters:	None.
Return:	The error code.

2 I/O Functions

2.1 Configuration

2.1.1 Set COM-Port Parameters

This function can be used to configure the asynchronous serial communication ports COM1 and COM2.

```
bool SetCOMParameters
    (
    int Channel,
    TCOMParameters* COMParameters
    );
```

Parameters:

`int Channel`	Specify the communication port which shall be configured. Use one of the predefined constants `CHANNEL_COM1` or `CHANNEL_COM2`.
`TCOMParameters* COMParameters`	Reference to the structure that holds the communication parameters. See the description of `TCOMParameters` for details.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
`unsigned long BaudRate`	32	This member holds the baud rate.
`byte WordLength`	8	This member holds the word-length in bits. Use the predefined constant `COM_WORDLENGTH_8`.
`byte Parity`	8	This member holds the type of parity to be used. Use one of the predefined constants `COM_PARITY_NONE`, `COM_PARITY_ODD` or `COM_PARITY_EVEN`.
`byte StopBits`	8	This member holds the number of stop bits. Use one of the predefined constants `COM_STOPBITS_0_5`, `COM_STOPBITS_1`, `COM_STOPBITS_1_5` or `COM_STOPBITS_2`.
`byte FlowControl`	8	This member holds the type of flow control to be used. Use the predefined constant `COM_FLOWCONTROL_NONE`.

Table 2.1: Definition of `TCOMParameters`

2.1.2 Get USB Device State

This function returns the functional state of the USB-controller in case the reader is running as USB-device.

```
int GetUSBDeviceState(void);
```

Parameters:	None.
Return:	Depending on the functional state, the return value is one of the predefined constants `USB_DEVICE_STATE_DEFAULT`, `USB_DEVICE_STATE_ADDRESSED`, `USB_DEVICE_STATE_CONFIGURED` or `USB_DEVICE_STATE_SUSPENDED`.

2.1.3 Get Host Channel

This function returns the channel, which is actually configured for host communication.

```
int GetHostChannel(void);
```

Parameters:	None.
Return:	The return value is one of the predefined constants `CHANNEL_NONE`, `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2` or `CHANNEL_I2C`.

2.2 Miscellaneous Functions

2.2.1 Wake Up Host

This function allows to remotely wake up a host, which is connected via USB. This function is supported by USB keyboard only.

```
void USBRemoteWakeup(void);
```

Parameters:	None.
Return:	None.

2.3 Data I/O

2.3.1 Query I/O Buffer Size

Use this function to retrieve the input/output buffer size of a specific communication channel.

```
int GetBufferSize
    (
    int Channel,
    int Dir
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL`, `CHANNEL_I2C` or `CHANNEL_RNG`.
`int Dir`	Specify the direction. Use one of the predefined constants `DIR_OUT` or `DIR_IN`.
Return:	The buffer size in bytes.

2.3.2 Get I/O Buffer Byte Count

Use this function to retrieve the number of bytes that are actually stored in the respective I/O buffer. In case of querying the output direction, the functions returns the number of bytes that have not been sent yet, in case of the input direction the number of available bytes that can be read is returned.

```
int GetByteCount
    (
    int Channel,
    int Dir
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL`, `CHANNEL_I2C` or `CHANNEL_RNG`.
`int Dir`	Specify the direction. Use one of the predefined constants `DIR_OUT` or `DIR_IN`.
Return:	The number of bytes that are stored in the buffer.

2.3.3 Test Empty

Check if there are any bytes in the specified I/O buffer.

```
bool TestEmpty
    (
    int Channel,
    int Dir
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL`, `CHANNEL_I2C` or `CHANNEL_RNG`.
`int Dir`	Specify the direction. Use one of the predefined constants `DIR_OUT` or `DIR_IN`.
Return:	If the buffer is empty, the return value is `true`, otherwise it is `false`.

2.3.4 Test Full

Check if the specified I/O buffer can receive any further data.

```
bool TestFull
    (
    int Channel,
    int Dir
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL`, `CHANNEL_I2C` or `CHANNEL_RNG`.
`int Dir`	Specify the direction. Use one of the predefined constants `DIR_OUT` or `DIR_IN`.
Return:	If the buffer is full, the return value is `true`, otherwise it is `false`.

2.3.5 Send Byte

Use this function to send one byte through a specific communication channel. If the respective output buffer is completely occupied, the function blocks until there is enough space.

```
void WriteByte
    (
    int Channel,
    byte Byte
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL` or `CHANNEL_I2C`.
`byte Byte`	The byte to be sent.
Return:	None.

2.3.6 Send Multiple Bytes

Use this function to send multiple bytes through a specific communication channel. If there is not enough space in the respective output buffer, the function sends the number of bytes that fit into the buffer and returns this value.

```
int WriteBytes
    (
    int Channel,
    const byte* Bytes,
    int ByteCount
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL` or `CHANNEL_I2C`.
`const byte* Bytes`	The bytes to be sent.
Return:	Number of bytes sent.

2.3.7 Read Byte

Use this function to read a byte from the input buffer of a specific communication channel. If there is no byte available, the function blocks until there is one.

```
byte ReadByte
    (
    int Channel
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL`, `CHANNEL_I2C` or `CHANNEL_RNG`.
Return:	The byte which was read from the input buffer.

2.3.8 Read Multiple Bytes

Use this function to read a desired number of bytes from the input buffer of a specific communication channel. If there is less data available than desired, the function reads the available number of bytes.

```
int ReadBytes
    (
    int Channel,
    byte* Bytes,
    int ByteCount
    );
```

Parameters:

`int Channel`	Specify the communication channel. Use one of the predefined constants `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2`, `CHANNEL_CCID_DATA`, `CHANNEL_CCID_CTRL`, `CHANNEL_I2C` or `CHANNEL_RNG`.
`byte* Bytes`	The received data is stored in this buffer.
`int ByteCount`	Specify the number of bytes to be read.
Return:	The byte which was read from the input buffer.

3 Memory Functions

3.1 Byte Operations

3.1.1 Compare Bytes

Compare two byte arrays.

```
bool CompBytes
    (
    const byte* Data1,
    const byte* Data2,
    int ByteCount
    );
```

Parameters:	
`const byte* Data1`	Reference to an array of bytes.
`const byte* Data1`	Reference to an array of bytes.
`int ByteCount`	Number of bytes (beginning from index 0) to be compared.
Return:	If the two arrays are identical, the return value is `true`, otherwise it is `false`.

3.1.2 Copy Bytes

Copy bytes from a source to a destination. Source and destination may be identical and the source section may overlap the destination. Depending on that, the correct method for copying will be chosen.

```
void CopyBytes
    (
    byte* DestBytes,
    const byte* SourceBytes,
    int ByteCount
    );
```

Parameters:	
`byte* DestBytes`	Reference to an array of bytes which is the destination of the copy operation.
`const byte* SourceBytes`	Reference to an array of bytes which is the source of the copy operation.
`int ByteCount`	Number of bytes to be copied.
Return:	None.

3.1.3 Fill Bytes

Fill bytes within a given array with a value.

```
void FillBytes
    (
    byte* Dest,
    byte Value,
    int ByteCount
    );
```

Parameters:	
`byte* Dest`	Reference to an array of bytes which is the destination for the operation.
`byte Value`	The byte value with which the array will be filled.
`int ByteCount`	Number of bytes to be filled.
Return:	None.

3.1.4 Swap Bytes

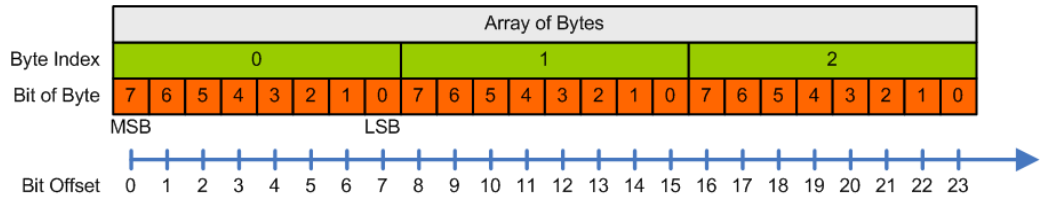
Swap the order of bytes within an array.

```
void SwapBytes
    (
    byte* Data,
    int ByteCount
    );
```

Parameters:	
`byte* Data`	Reference to an array of bytes which is the destination for the operation.
`int ByteCount`	Number of bytes to be swapped.
Return:	None.

3.2 Bit Operations

Bit operations are working on bit fields. A bit field is represented by an array of bytes. The diagram below shows how bit operations are interpreting a given bit offset within an array of bytes:



3.2.1 Read Bit

Read the value of one single bit within a bit field.

```
bool ReadBit
    (
    const byte* Byte,
    int BitNr
    );
```

Parameters:

`const byte* Byte`	Reference to an array of bytes which represents the bit field where one bit shall be read.
`int BitNr`	Position of the bit within the bit field.
Return:	The bit value: `true` means 1, `false` means 0.

3.2.2 Write Bit

Set one single bit within a bit field to a given value.

```
void WriteBit
    (
    byte* Byte,
    int BitNr,
    bool Value
    );
```

Parameters:

`byte* Byte`	Reference to an array of bytes which represents the bit field where one bit shall be written.
`int BitNr`	Position within the bit field, where the bit is to be written.
`bool Value`	The bit value: `true` means 1, `false` means 0.
Return:	None.

3.2.3 Copy Bit

Copy one single bit from a source to a destination. Source and destination may be identical.

```
void CopyBit
    (
    byte* Dest,
    int DestBitNr,
    const byte* Source,
    int SourceBitNr
    );
```

Parameters:	
`byte* Dest`	Reference to an array of bytes which is the destination for the operation.
`int DestBitNr`	Position within the destination bit field, where the bit is copied to.
`const byte* Source`	Reference to an array of bytes which is the source for the operation.
`int SourceBitNr`	Position within the source bit field, where the bit is copied from.
Return:	None.

3.2.4 Compare Bits

Compare two bit sets.

```
bool CompBits
    (
    const byte* Data1,
    int Data1StartBit,
    const byte* Data2,
    int Data2StartBit,
    int BitCount
    );
```

Parameters:	
`const byte* Data1`	Reference to an array of bytes which represents a bit field.
`int Data1StartBit`	Start-index (beginning from 0) of the first bit field.
`const byte* Data2`	Reference to an array of bytes which represents a bit field.
`int Data1StartBit`	Start-index (beginning from 0) of the second bit field.
`int BitCount`	Number of bits to be compared.
Return:	If the two bit-sets are identical, the return value is `true`, otherwise it is `false`.

3.2.5 Copy Bits

Copy bits from a source to a destination. Source and destination may be identical and the source section may overlap the destination. Depending on that, the correct method for copying will be chosen.

```
void CopyBits
    (
    byte* DestBits,
    int StartDestBit,
    const byte* SourceBits,
    int StartSourceBit,
    int BitCount
    );
```

Parameters:	
`byte* DestBits`	Reference to an array of bytes which represents a bit field which is the destination of the copy operation.
`int StartDestBit`	First bit within the destination bit field where the bits are copied to.
`const byte* SourceBits`	Reference to an array of bytes which represents a bit field which is the source of the copy operation.
`int StartSourceBit`	First bit within the source bit field where the bits are copied from.
`int BitCount`	Number of bits to be copied.
Return:	None.

3.2.6 Fill Bits

Fill bits within a given bit field with either 0 or 1.

```
void FillBits
    (
    byte* Dest,
    int StartBit,
    bool Value,
    int BitCount
    );
```

Parameters:	
`byte* Dest`	Reference to an array of bytes which represents a bit field which is the destination for the operation.
`int StartBit`	First bit within the bit field where the bits are filled.
`bool Value`	The bit value: `true` means 1, `false` means 0.
`int BitCount`	Number of bits to be filled.
Return:	None.

3.2.7 Swap Bits

Swap the order of bits within a bit field.

```
void SwapBits
    (
    byte* Data,
    int StartBit,
    int BitCount
    );
```

Parameters:

`byte* Data`	Reference to an array of bytes which represents a bit field which is the destination for the operation.
`int StartBit`	First bit within the bit field where bits are swapped.
`int BitCount`	Number of bits to be swapped.
Return:	None.

3.2.8 Count Bits

Count the number of ones or zeros within a bit field.

```
int CountBits
    (
    const byte* Data,
    int StartBit,
    bool Value,
    int BitCount
    );
```

Parameters:

`const byte* Data`	Reference to an array of bytes which represents a bit field.
`int StartBit`	First bit within the bit field where counting shall start.
`bool Value`	The bit value: `true` means count ones, `false` means count zeros.
`int BitCount`	Size of the bit field.
Return:	Number of counted bits.

4 Peripheral Functions

4.1 General Purpose Inputs/Outputs (GPIOs)

4.1.1 Configuration

4.1.1.1 Outputs

Use this function to configure one or several GPIOs as output. Each output can be configured to have an integrated pull-up or pull-down resistor. The output driver characteristic is either Push-Pull or Open Drain.

```
void GPIOConfigureOutputs
    (
    int Bits,
    int PullUpDown,
    int OutputType
    );
```

Parameters:

`int Bits`	Specify the GPIOs that shall be configured for output. Several GPIOs can be configured simultaneously by using the bitwise or-operator (|). Use the predefined constants `GPIO0` through `GPIO7` for specifying the GPIOs.
`int PullUpDown`	Specify the behaviour of the internal weak pull-up/down resistor. Use one of the predefined constants `GPIO_PUPD_NOPULL`, `GPIO_PUPD_PULLUP` or `GPIO_PUPD_PULLDOWN`.
`int OutputType`	Specify the output driver characteristic. Use one the predefined constants `GPIO_OTYPE_PUSHPULL` or `GPIO_OTYPE_OPENDRAIN`.
Return:	None.

4.1.1.2 Inputs

Use this function to configure one or several GPIOs as input. Each output can be configured to have an integrated pull-up or pull-down resistor, alternatively it can be left floating.

```
void GPIOConfigureInputs
    (
    int Bits,
    int PullUpDown
    );
```

Parameters:

`int Bits`	Specify the GPIOs that shall be configured for input. Several GPIOs can be configured simultaneously by using the bitwise or-operator (|). Use the predefined constants `GPIO0` through `GPIO7` for specifying the GPIOs.
`int PullUpDown`	Specify the behaviour of the internal weak pull-up/down resistor. Use one of the predefined constants `GPIO_PUPD_NOPULL`, `GPIO_PUPD_PULLUP` or `GPIO_PUPD_PULLDOWN`.
Return:	None.

4.1.2 Basic Port Functions

4.1.2.1 Set GPIOs to Logical Level

Use this function to set one or several GPIOs to logical high or low level. The respective ports must have been configured to output in advance.

```
void GPIOSetBits(int Bits);
void GPIOClearBits(int Bits);
```

Parameters:

`int Bits`	Specify the GPIOs that shall be set to a logical level. Several GPIOs can be handled simultaneously by using the bitwise or-operator (|). Use the predefined constants `GPIO0` through `GPIO7` for specifying the GPIOs.
Return:	None.

4.1.2.2 Toggle GPIOs

Use this function to toggle the logical level of one or several GPIOs. The respective ports must have been configured to output in advance.

```
void GPIOToggleBits
    (
    int Bits
    );
```

Parameters:

`int Bits`	Specify the GPIOs that shall be toggled. Several GPIOs can be handled simultaneously by using the bitwise or-operator (|). Use the predefined constants `GPIO0` through `GPIO7` for specifying the GPIOs.
Return:	None.

4.1.2.3 Waveform Generation

Use this function to generate a pulse-width modulated square waveform with constant frequency on one or several GPIOs. The respective ports must have been configured to output in advance.

```
void GPIOBlinkBits
    (
    int Bits,
    int TimeHi,
    int TimeLo
    );
```

Parameters:

`int Bits`	Specify the GPIOs that shall generate the waveform. Several GPIOs can be handled simultaneously by using the bitwise or-operator (|). Use the predefined constants `GPIO0` through `GPIO7` for specifying the GPIOs.
`int TimeHi`	Specify the duration for logical high level in milliseconds.
`int TimeLo`	Specify the duration for logical low level in milliseconds.
Return:	None.

4.1.2.4 Read GPIOs

Use this function to read the logical level of one GPIO that has been configured as input.

```
int GPIOTestBit
    (
    int Bit
    );
```

Parameters:

`int Bits`	Specify the GPIO that shall be read. Use one of the predefined constants `GPIO0` through `GPIO7` for specifying the GPIO.
Return:	If the GPIO has logical high level, the return value is 1, otherwise it is 0.

4.1.3 Higher Level Port Functions

4.1.3.1 Send Data in Wiegand Format

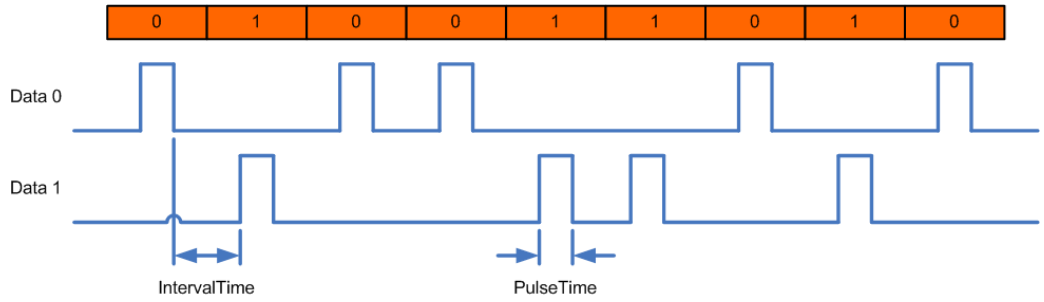
Use this function to send a bitstream via a software emulated Wiegand interface. A Wiegand interface uses two data lines, one line is used to transmit ones, the other one is used to transmit zeros. Each GPIO can be individually configured to act as data line. Note that the integrated API LED-functions are working with GPIO0 to GPIO2 by default, so the Wiegand data lines should be selected carefully.

```
void SendWiegand(int GPIOData0,int GPIOData1,int PulseTime,
                 int IntervalTime,byte* Bits,int BitCount);
```

Parameters:

`int GPIOData0`	Specify the GPIO that shall be used to transmit zeros. Use one of the predefined constants `GPIO0` through `GPIO7` for specifying the GPIO.
`int GPIOData1`	Specify the GPIO that shall be used to transmit ones. Use one of the predefined constants `GPIO0` through `GPIO7` for specifying the GPIO.
`int PulseTime`	Specify the pulse duration in microseconds.
`int IntervalTime`	Specify the duration in microseconds between consecutive pulses.
`byte* Bits`	Reference to an array of bytes which represents a bit field which holds the data to be sent.
`int BitCount`	Specify the number of bits to be sent.
Return:	None.

See timing diagram below for details about how the timing values are used:



Example:

Here is an example which shows minimum code for doing a Wiegand output:

```
// Init Section:
// Use GPIO2 and GPIO3 for Wiegand interface
GPIOConfigureOutputs(GPIO2 | GPIO3,GPIO_PUPD_NOPULL,GPIO_OTYPE_PUSHPULL);
// Enter idle level. In this case we have active low outputs
GPIOSetBits(GPIO2 | GPIO3);
// Prepare some Wiegand data:
byte Bits[4];
Bits[0] = 0x12;
Bits[1] = 0x34;
Bits[2] = 0x56;
Bits[3] = 0x78;
// Now send the bits
SendWiegand(GPIO2,GPIO3,100,1000,Bits,32);
```

Note:

- It is up to the App to complete Wiegand data with parity bits and decide number of bits. In this way the App is fully flexible regarding data to be sent.
- The idle level of the Wiegand interface is determined by state of the outputs before calling `SendWiegand`. It must be setup by a separate call to `GPIOSetBits` or `GPIOClearBits` depending on the requirements of the underlying hardware.
- The GPIOs might need additional circuitry against shortcut or voltage level depending on the intended application.

4.1.3.2 Send Data in Omron Format

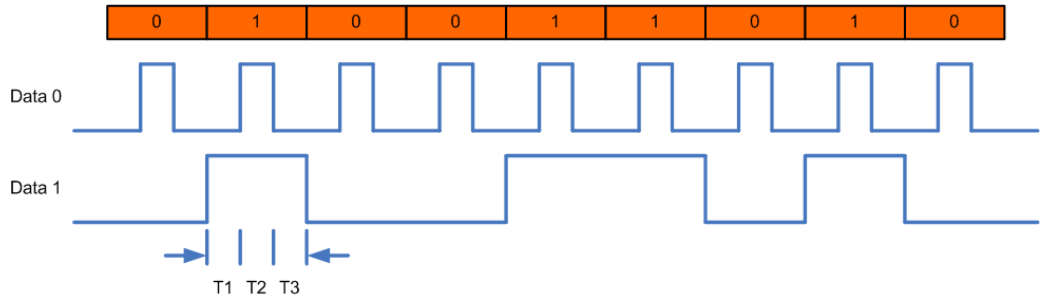
Use this function to send a bit stream via a software-emulated Omron interface. An Omron interface uses two lines for data transmission, one for clock and one for the data bit stream. Each GPIO can be individually configured to act as data or clock line. Note that the integrated API LED-functions are working with GPIO0 to GPIO2 by default, so the Omron interface lines should be selected carefully.

```
void SendOmron(int GPIOClock,int GPIOData,int T1,int T2,int T3,
               byte* Bits,int BitCount);
```

Parameters:

`int GPIOClock`	Specify the GPIO that shall be used for generating the clock signal. Use one of the predefined constants `GPIO0` through `GPIO7` for specifying the GPIO.
`int GPIOData`	Specify the GPIO that shall be used for data transmission. Use one of the predefined constants `GPIO0` through `GPIO7` for specifying the GPIO.
`int T1`	
`int T2`	
`int T3`	
`byte* Bits`	Reference to an array of bytes which represents a bit field which holds the data to be sent.
`int BitCount`	Specify the number of bits to be sent.
Return:	None.

See timing diagram below for details about how the timing values are used:



Example:

Here is an example which shows minimum code for doing a clock/data output:

```
// Init Section:
// Use GPIO2 and GPIO3 for the clock/data interface
GPIOConfigureOutputs(GPIO2 | GPIO3,GPIO_PUPD_NOPULL,GPIO_OTYPE_PUSHPULL);
// Enter idle level. In this case we have active low outputs
GPIOSetBits(GPIO2 | GPIO3);
// Prepare some data:
byte Bits[4];
Bits[0] = 0x12;
Bits[1] = 0x34;
Bits[2] = 0x56;
Bits[3] = 0x78;
// Now send the bits
SendOmron(GPIO2,GPIO3,500,1000,500,Bits,32);
```

Note:

- It is up to the App to complete data with parity bits and decide number of bits. In this way the App is fully flexible regarding data to be sent.
- The idle level of the clock/data interface is determined by state of the outputs before calling `SendOmron`. It must be setup by a separate call to `GPIOSetBits` or `GPIOClearBits` depending on the requirements of the underlying hardware.
- The GPIOs might need additional circuitry against shortcut or voltage level depending on the intended application.

4.2 Beeper

Use this function to sound a beep at the dedicated beeper port.

```
void Beep
    (
    int Volume,
    int Frequency,
    int OnTime,
    int OffTime
    );
```

Parameters:	
`int Volume`	Specify the volume in percent from 0 to 100.
`int Frequency`	Specify the frequency in Hertz.
`int OnTime`	Specify the duration of the beep in milliseconds.
`int OffTime`	Specify the length of the pause after the beep. This is useful for generating melodies. If this is not required, the parameter may have the value 0.
Return:	None.

4.3 LEDs

4.3.1 General Purpose LED Functions

These functions are related for usage with TWN4 Desktop and TWN4 Panel where the different LEDs have a dedicated connection scheme. The LEDs are connected as follows:

- GPIO0 → Red
- GPIO1 → Green
- GPIO2 → Yellow (Panel version only)

4.3.1.1 Initialization

Use this macro to initialize the respective GPIOs to drive LEDs.

```
LEDInit(LEDs);
```

Parameters:	
LEDs	Specify the GPIOs that shall be configured for LED operation. Several GPIOs can be configured simultaneously by using the bitwise or-operator (|). Use the predefined constants REDLED, GREENLED or YELLOWLED for specifying the GPIOs.
Return:	None.

4.3.1.2 Set LEDs On/Off

Use these macros to set one or several LEDs on/off.

```
LEDOn(LEDs);
LEDOff(LEDs);
```

Parameters:	
LEDs	Specify the LEDs that shall be set on/off. Several LEDs can be handled simultaneously by using the bitwise or-operator (|). Use the predefined constants REDLED, GREENLED or YELLOWLED for specifying the LEDs.
Return:	None.

4.3.1.3 Toggle LEDs

Use this macro to toggle one or several LEDs.

```
LEDToggle(LEDs);
```

Parameters:	
LEDs	Specify the LEDs that shall be toggled. Several LEDs can be handled simultaneously by using the bitwise or-operator (|). Use the predefined constants REDLED, GREENLED or YELLOWLED for specifying the LEDs.
Return:	None.

4.3.1.4 Blink LEDs

Use this macro to let one or several LEDs blink.

```
LEDBlink(LEDs, TimeOn, TimeOff);
```

Parameters:	
LEDs	Specify the LEDs that shall blink. Several LEDs can be handled simultaneously by using the bitwise or-operator (|). Use the predefined constants REDLED, GREENLED or YELLOWLED for specifying the LEDs.
TimeOn	Specify the on-time in milliseconds.
TimeOff	Specify the off-time in milliseconds.
Return:	None.

4.3.1.5 Get LED State

Use this macro to determine if a LED is on or off.

```
LEDIsOn(LED);
```

Parameters:	
LED	Specify the LED that shall be queried. Use one of the predefined constants REDLED, GREENLED or YELLOWLED for specifying the LED.
Return:	If the queried LED is on, the return value is 1, otherwise it is 0.

4.3.2 Diagnostic LED

The TWN4 Core Module has one integrated LED that can be used for diagnostic purposes. There is no initialization necessary.

4.3.2.1 Set Diagnostic LED On/Off

Use these functions to set the diagnostic LED on or off.

```
void DiagLEDOn(void);
void DiagLEDOff(void);
```

Parameters:	None.
Return:	None.

4.3.2.2 Toggle Diagnostic LED

Use this function to toggle the diagnostic LED.

```
void DiagLEDToggle(void);
```

Parameters:	None.
Return:	None.

4.3.2.3 Get LED State

Use this function to determine if the diagnostic LED is on or off.

```
bool DiagLEDIsOn(void);
```

Parameters:	None.
Return:	If the diagnostic LED is on, the return value is `true`, otherwise it is `false`.

5 Conversion Functions

5.1 Hexadecimal ASCII to Binary

5.1.1 Scan Hexadecimal Character

Convert an ASCII-character which represents a hexadecimal number into its binary representation.

```
int ScanHexChar
    (
    byte Char
    );
```

Parameters:	
`byte Char`	ASCII-coded hexadecimal character. The input value may be one of the characters '0'-'9', 'a'-'f' or 'A'-'F'.
Return:	If the character is a valid hexadecimal expression, the return value is the binary representation (a number between 0 and 15), else it is -1.

5.1.2 Scan Hexadecimal String

Convert an array of bytes containing ASCII characters which represents hexadecimal numbers into their binary representation. The conversion is done in place. This means that after successful conversion, number of valid bytes is half of the given count of ASCII characters (two hex digits represent one binary byte).

```
int ScanHexString
    (
    byte* ASCII,
    int ByteCount
    );
```

Parameters:	
`byte* ASCII`	Reference to an array of ASCII-coded hexadecimal characters. The array may contain the characters '0'-'9', 'a'-'f' or 'A'-'F'. The array is also the destination for the operation.
`int ByteCount`	Number of (ASCII-) bytes to be converted.
Return:	Number of successfully converted bytes.

5.2 Binary to Hexadecimal ASCII

Convert a number, which is given as a bit field into ASCII format, and store it in an array of bytes. The conversion is made in the following sequence:

1. Convert the binary data to a number of digits, which is determined by the parameter `MaxDigits`. If `MaxDigits` is 0, then the number of digits is determined by the binary data itself.
2. If the result of the conversion is less than the number of digits specified by `MinDigits`, precede the converted number with zeros according to `MinDigits`.

```
int ConvertBinaryToString
    (
    const byte* SourceBits,
    int StartBit,
    int BitCnt,
    char* String,
    int Radix,
    int MinDigits,
    int MaxDigits
    );
```

Parameters:

`const byte* SourceBits`	A reference to an array of bytes, which contains the bit field.
`int StartBit`	Index of the first bit to be converted.
`int BitCnt`	The number of bits, which are valid within the array of bytes.
`char* String`	A reference to an array of bytes, which receives the result of the conversion.
`int Radix`	Base for conversion, use: • 2 for binary conversion • 8 for octal conversion • 10 for decimal conversion • 16 for hexadecimal conversion
`int MinDigits`	Specifies the minimum number of digits, the output should contain. If `MinDigits` is 0, then at least 1 digit is sent. If `MinDigits` is greater than the actual width of the number to be converted, then the number is preceded by zeros.
`int MaxDigits`	Specifies the maximum number of digits, the output may contain. `MaxDigits` has higher priority than `MinDigits`.
Return:	The actual number of ASCII bytes, which has been stored in the array `String`.

6 I2C Functions

This chapter describes functions for accessing the I2C interface of TWN4. I2C is also known as TWI (Two-Wire Interface).

6.1 Initialization/Deinitialization

6.1.1 I2CInit

```
bool I2CInit(int Mode);
```

Parameters:	
`int Mode`	This value specifies the mode of operation.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

6.1.2 I2CDeInit

```
void I2CDeInit(void);
```

Parameters:	None.
Return:	None.

6.1.3 Examples

```
// Initialize as master
I2CInit(I2CMODE_MASTER);

// Initialize as slave.
//   I2CMODE_SLAVE: Setup interface as slave
//            0x30: Address of of this slave
// I2CMODE_CHANNEL: Do communication via channels (this is the
//                  only currently available option, therefore
//                  a must to be specified)
I2CInit(I2CMODE_SLAVE | 0x30 | I2CMODE_CHANNEL);
```

6.2 Communication (Master)

6.2.1 I2CMasterStart

Generate a I2C start sequence.

```
void I2CMasterStart(void);
```

Parameters:	None.
Return:	None.

6.2.2 I2CMasterStop

Generate a I2C stop sequence.

```
void I2CMasterStop(void);
```

Parameters:	None.
Return:	None.

6.2.3 I2CMasterTransmitByte

Transmit one byte to a slave.

```
void I2CMasterTransmitByte(byte Byte);
```

Parameters:	
`byte Byte`	The byte to be transmitted to the slave.
Return:	None.

6.2.4 I2CMasterReceiveByte

Receive one byte from a slave.

```
byte I2CMasterReceiveByte(void);
```

Parameters:	None.
Return:	The byte read from the slave.

6.2.5 I2CMasterBeginWrite

Begin a write sequence. This will send the target slave address together with R/W-bit set to write.

```
void I2CMasterBeginWrite(int Address);
```

Parameters:	
`int Address`	The target slave address, a value from 0 to 127.
Return:	None.

6.2.6 I2CMasterBeginRead

Begin a read sequence. This will send the target slave address together with R/W-bit set to read.

```
void I2CMasterBeginRead(int Address);
```

Parameters:	
`int Address`	The target slave address, a value from 0 to 127.
Return:	None.

6.2.7 I2CMasterSetAck

Set ACK state of the master. This ACK will be sent after receiption of one byte from the slave.

```
void I2CMasterSetAck(bool SetOn);
```

Parameters:	
`bool SetOn`	Set this value to `true` to turn acknowledge on or `false` to turn acknowledge off. Definitions `ON` or `OFF` may be used for better readability.
Return:	None.

6.2.8 Examples

```
// This sample demonstrates transmission and receiption of data
// to/from a I2C-slave

// This is the address of the slave
const int I2CAddress = 0x30;
// Init the I2C port
I2CInit(I2CMODE_MASTER);

// Send two bytes to the slave
I2CMasterStart();
I2CMasterBeginWrite(I2CAddress);
I2CMasterTransmitByte(0x12);
I2CMasterTransmitByte(0x34);
I2CMasterStop();

// Receive three bytes from the slave
byte Bytes[3];
I2CMasterStart();
I2CMasterBeginRead(I2CAddress);
// All bytes except last byte require an ACK to be sent
I2CMasterSetAck(ON);
Bytes[0] = I2CMasterReceiveByte();
Bytes[1] = I2CMasterReceiveByte();
// Turn off ACK before reading last byte
I2CMasterSetAck(OFF);
Bytes[2] = I2CMasterReceiveByte();
I2CMasterStop();
```

6.3 Communication (Slave)

Communication as a I2C slaves works with well-defined I2C packets, which must be sent between master and slave (TWN4).

The communication is performed via normal communication channels. Therefore, for transmitting and receiving data, the normal IO-functions must be used. These are `WriteByte`, `ReadByte` and so on. In case of communication via I2C, the channel 4 must be used. There is a definition for this channel, which is `CHANNEL_I2C`.

As a conclusion, TWN4 offers a easy method of changing communication from USB or RS232 to I2C just by changing the communication channel. Only care must be taken to avoid buffer overflow. This can be

achieved by calling appropriate IO-functions `TestEmpty` and `TestFull`. On the other hand many communication protocols avoid a buffer overflow by their inherent flow of communication (e.g. command/response protocol).

The specification for the format of the packets sent/reveived on the I2C bus is as follows:

6.3.1 Slave to Master

`1 Byte`	Address/Read
`1 Byte`	Buffer status: Bits 7..4 hold the number of bytes, which are available to be read from the slave. Bits 3..0 hold the maximum number of bytes, which may be sent to slave.
`n Bytes`	Payload, where n is 0..15. Note: Due to the fact, that ACK must be turned off one byte before the master receives last byte, it is useful to check buffer status and receive bytes in separate read operations.

6.3.2 Master to Slave

`1 Byte`	Address/Write
`n Bytes`	Payload, where n is 1..15

6.3.3 Examples

This is a implementation of a I2C master communication, which routes USB- or RS232-interface to the I2C-interface of a TWN4 Core Module. In order to test this example, two TWN4 Core Modules are required:

- 1 TWN4 Core Module, which is running as I2C slave
- 1 TWN4 Core Module, which is running as I2C master.

```
//
// TWN4 App: I2C master, which routes USB or RS232-traffic to I2C
//
#include "twn4.sys.h"
#include "apptools.h"

int main(void)
{
    const int I2CAddress = 0x30;
    // USB or RS232 depends on which cable is connected
    int HostChannel = GetHostChannel();

    I2CInit(I2CMODE_MASTER);
    while (true)
    {
        int I2CRXTXCount;
        int TransferCount;

        I2CMasterStart();
        I2CMasterBeginRead(I2CAddress);
        I2CMasterSetAck(OFF);
```

```
        I2CRXTXCount = I2CMasterReceiveByte();
        I2CMasterStop();

        // ************************************************************
        // ****** Direction Host -> I2C *****************************
        // ************************************************************
        TransferCount = MIN(GetByteCount(HostChannel,DIR_IN),
                            I2CRXTXCount & 0x0F);
        if (TransferCount > 0)
        {
            I2CMasterStart();
            I2CMasterBeginWrite(I2CAddress);
            while (TransferCount-- > 0)
                I2CMasterTransmitByte(ReadByte(HostChannel));
            I2CMasterStop();
        }

        // ************************************************************
        // ****** Direction I2C -> Host *****************************
        // ************************************************************
        TransferCount = MIN(GetBufferSize(HostChannel,DIR_OUT)-
                            GetByteCount(HostChannel,DIR_OUT),
                            I2CRXTXCount >> 4);
        if (TransferCount > 0)
        {
            I2CMasterStart();
            I2CMasterBeginRead(I2CAddress);
            I2CMasterSetAck(ON);
            // Flush RX/TX byte count
            I2CMasterReceiveByte();
            // Read data except last byte
            while (TransferCount-- > 1)
                WriteByte(HostChannel,I2CMasterReceiveByte());
            // Turn off ACK before reading last byte
            I2CMasterSetAck(OFF);
            WriteByte(HostChannel,I2CMasterReceiveByte());
            I2CMasterStop();
        }
    }
}
```

7 SPI Functions

This chapter describes functions for accessing the SPI interface of TWN4. Currently, the SPI interface can be operated as master.

7.1 Initialization/Deinitialization

7.1.1 SPIInit

Initialize communication via SPI interface.

```
bool SPIInit(const TSPIParameters *SPIParameters);
```

Parameters:

`const TSPIParameters *SPIParameters`	Pointer to a structure, which specifies mode of operation.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

The members of structure `TSPIParameters` are defined as follows:

Members	Length (Bits)	Description
`byte Mode`	8	Mode of operation. Please always specify `SPI_MODE_MASTER` here.
`byte CPOL`	8	Polarity if clock signal (SPI_SCK). Specify `SPI_CPOL_LOW` for idle/inactive low or `SPI_CPOL_HIGH` for idle/inactive high.
`byte CPHA`	8	Active edge of SPI_SCK. Specify `SPI_CPHA_EDGE1` for first edge or `SPI_CPHA_EDGE2` for second edge. In conjunction with the polarity of the clock signal this leads to active edge, which is either rising or falling.
`byte ClockRate`	8	Specify clock rate of SPI_SCK. Valid values are `SPI_CLOCKRATE_117_KHZ`, `SPI_CLOCKRATE_234_KHZ`, `SPI_CLOCKRATE_469_KHZ`, `SPI_CLOCKRATE_938_KHZ`, `SPI_CLOCKRATE_1_88_MHZ`, `SPI_CLOCKRATE_3_75_MHZ`, `SPI_CLOCKRATE_7_5_MHZ` or `SPI_CLOCKRATE_15_MHZ`
`byte BitOrder`	8	Specify order of data bits on data lines (SPI_MISO and SPI_MOSI). Specify `SPI_FIRSTBIT_MSB` for idle/inactive low or `SPI_FIRSTBIT_LSB` for idle/inactive high.

Table 7.1: Definition of `TSPIParameters`

7.1.2 SPIDeInit

Deinitialize SPI interface.

```
void SPIDeInit(void);
```

Parameters:	None.
Return:	None.

7.2 Communication

7.2.1 SPIMasterBeginTransfer

Begin a transfer via SPI interface. This function sets signal SPI_SS- to active, thus low.

```
void SPIMasterBeginTransfer(void);
```

Parameters:	None.
Return:	None.

7.2.2 SPIMasterEndTransfer

End a transfer via SPI interface. This function sets signal SPI_SS- to inactive, thus high.

```
void SPIMasterEndTransfer(void);
```

Parameters:	None.
Return:	None.

7.2.3 SPITransceive

Send and receive a number of bytes to/from the slave. Background: SPI is a full duplex communication link. This allows to send and receive data at the same time. With every clock pulse, a bit is sent to the slave, another bit is received from the slave.

```
bool SPITransceive(const byte *TXData,byte *RXData,int ByteCount);
```

Parameters:	
`const byte *TXData`	Pointer to an array of bytes being transmitted to the slave.
`byte *RXData`	Pointer to an array of bytes being received from the slave.
`int ByteCount`	Number of bytes transferred in each direction.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

7.2.4 SPITransmit

Send a number of bytes to the slave. Received bits are refused.

```
bool SPITransmit(const byte *TXData,int ByteCount);
```

Parameters:	
`const byte *TXData`	Pointer to an array of bytes being transmitted to the slave.
`int ByteCount`	Number of bytes transmitted.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

7.2.5 SPIReceive

Receive a number of bytes from the slave. Transmitted bits are set to zero.

```
bool SPIReceive(byte *RXData,int ByteCount);
```

Parameters:	
`byte *RXData`	Pointer to an array of bytes being received from the slave.
`int ByteCount`	Number of bytes received.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

7.3 Examples

```
#include "twn4.sys.h"

void FuncSPIInitMaster(void)
{
    const TSPIParameters Mode =
    {
        SPI_MODE_MASTER,
        SPI_CPOL_LOW,
        SPI_CPHA_EDGE1,
        SPI_CLOCKRATE_15_MHZ,
        SPI_FIRSTBIT_MSB
    };
    SPIInit(&Mode);
}
void FuncSPITransmitPacket(void)
{
    SPIMasterBeginTransfer();
    static const byte TXData[4] = { 'A','B','C','D' };
    SPITransmit(TXData,4);
    SPIMasterEndTransfer();
}
void FuncSPIReceivePacket(void)
{
    SPIMasterBeginTransfer();
    byte RXData[4];
```

```
    SPIReceive(RXData,4);
    SPIMasterEndTransfer();
}
void FuncSPITransceivePacket(void)
{
    SPIMasterBeginTransfer();
    static const byte TXData[4] = { 'A','B','C','D' };
    byte RXData[4];
    SPITransceive(TXData,RXData,4);
    SPIMasterEndTransfer();
}
```

8 RF Functions

8.1 SearchTag

Use this function to search a transponder in the reading range of TWN4. TWN4 is searching for all types of transponders, which have been specified via function `SetTagTypes`. If a transponder has been found, tag type, length of ID and ID data itself are returned.

```
bool SearchTag(int *TagType,int *IDBitCount,byte *ID,int MaxIDBytes);
```

Parameters:	None.
`int *TagType`	Pointer to an integer, which receives the type of tag, which has been found.
`int *IDBitCount`	Pointer to an integer, which receives the number of bits(!), the ID consists of.
`byte *ID`	Pointer to an array of bytes, which contain ID data, if a transponder has been found.
`int MaxIDBytes`	A value, which specifies the buffer size of `ID`. No more than this specified number of bytes will be copied to the location specified by `ID`.
Return:	If a transponder has been found, the function returns `true`, otherwise it returns `false`.

8.2 SetRFOff

Turn off RF field. If no further operations are required on a transponder found via function `SearchTag` you may use this command to minimize power consumption of TWN4.

```
void SetRFOff(void);
```

Parameters:	None.
Return:	None.

8.3 SetTagTypes

Use this function to configure the transponders, which are searched by function `SearchTag`.

```
void SetTagTypes(unsigned int LFTagTypes,unsigned int HFTagTypes);
```

Parameters:	
`unsigned int LFTagTypes`	Specifies transponder types at the frequency 125.0 kHz - 134.2 kHz.
`unsigned int HFTagTypes`	Specifies transponder types at the frequency 13.56 MHz.
Return:	None.

8.3.1 Supported Types of LF Tags (125 kHz - 134.2 kHz)

Definition	Frequency	Name	Status
LFTAG_EM4102	LF	EM4102 / CASI-RUSCO	Supported
LFTAG_HITAG1S	LF	HITAG 1 / HITAG S	Supported
LFTAG_HITAG2	LF	HITAG 2	Supported
LFTAG_EM4150	LF	EM4x50	Supported
LFTAG_AT5555	LF	AT5555 / AT5557 / AT5577 / Q5	Supported, delivers no ID
LFTAG_ISOFDX	LF	ISO FDX-B / EM4105	Supported
LFTAG_EM4026	LF	EM4026	On request
LFTAG_HITAGU	LF	HITAG µ	On request
LFTAG_EM4305	LF	EM4305	On roadmap
LFTAG_HIDPROX	LF	HID Prox	Supported with option P
LFTAG_TIRIS	LF	ISO HDX / TIRIS	Supported
LFTAG_COTAG	LF	Cotag	Supported by option P
LFTAG_IOPROX	LF	ioProx	Supported by option P
LFTAG_INDITAG	LF	Indala	Supported by option P
LFTAG_HONEYTAG	LF	NexWatch	Supported by option P
LFTAG_AWID	LF	AWID	Supported
LFTAG_GPROX	LF	G-Prox	Supported, read of hash value
LFTAG_PYRAMID	LF	Pyramid	Supported
LFTAG_KERI	LF	Keri	Supported, read of raw data
LFTAG_DEISTER	LF	Deister	Supported, read of raw data
LFTAG_CARDAX	LF	Cardax	Supported, read of hash value
LFTAG_NEDAP	LF	Nedap	Supported, read of hash value
LFTAG_PAC	LF	PAC	Supported, read of hash value
LFTAG_IDTECK	LF	IDTECK	Supported, read of raw data

8.3.2 Supported Types of HF Tags (13.56 MHz, Bluetooth)

Definition	Frequency	Name	Status
HFTAG_MIFARE	HF	ISO14443A / MIFARE	Supported
HFTAG_ISO14443B	HF	ISO14443B	Supported
HFTAG_ISO15693	HF	ISO15693 / Tag-it	Supported
HFTAG_LEGIC	HF	LEGIC	Supported by TWN4 LEGIC
HFTAG_HIDICLASS	HF	HID iCLASS	Supported, read of UID, read of PAC with option I
HFTAG_FELICA	HF	FeliCa	Supported, read of UID only
HFTAG_SRX	HF	SRC	Supported
HFTAG_NFCP2P	HF	NFC Peer-to-Peer	Supported
HFTAG_BLE	HF	BLE (Bluetooth Low Energy)	Supported by TWN4 MultiTech 2 BLE
HFTAG_TOPAZ	HF	Topaz	Not supported by TWN4 LEGIC

In order to search for more than one type of transponder, several types can be combined.

Note:

The use of the predefined macro TAGMASK is mandatory, even if only one type of tag is specified. Here is an example which is searching for EM4102 and HITAG 1 at LF and for MIFARE at HF:

Example:

```
SetTagTypes(TAGMASK(LFTAG_EM4102) | TAGMASK(LFTAG_HITAG1S),
            TAGMASK(HFTAG_MIFARE));
```

8.4 GetTagTypes

This function returns the transponder types currently being searched for by function SearchTag separated by frequency (LF and HF).

```
void GetTagTypes(unsigned int *LFTagTypes,unsigned int *HFTagTypes);
```

Parameters:

`unsigned int *LFTagTypes` Pointer to a value, which receives the LF tag types.

`unsigned int *HFTagTypes` Pointer to a value, which receives the HF tag types.

Return: None.

8.5 GetSupportedTagTypes

This function returns the transponder types, which are actually supported by the individual TWN4 separated by frequency (LF and HF). Also the P-option is taken into account. This means, if the specific TWN4 has no option P, the appropriate transponders are not returned as supported type of transponder.

```
void GetSupportedTagTypes(unsigned int *LFTagTypes,
                          unsigned int *HFTagTypes);
```

Parameters:

`unsigned int *LFTagTypes`	Pointer to a value, which receives the LF tag types.
`unsigned int *HFTagTypes`	Pointer to a value, which receives the HF tag types.

Return: None.

9 EM4x02-Specific Transponder Operations

This chapter describes one function for accessing EM4x02 transponders. EM4x02 is a broadly known type of transponder, which is known under several names, such as EM4002, EM4102, Unique and several other brands.

9.1 Function

9.1.1 EM4102_GetTagInfo

Get detailed information regarding transponder type `LFTAG_EM4102` being found by function `SearchTag`.

```
int EM4102_GetTagInfo(void)
```

Parameters:	None.
Return:	One of the following pre-defined values: `EM4102_BITRATE_UNKNOWN` or `EM4102_BITRATE_F64` or `EM4102_BITRATE_F32`.

10 HITAG 1- and HITAG S-Specific Transponder Operations

This chapter describes functions for accessing HITAG 1 and HITAG S transponders. HITAG 1 and HITAG S are very similar. Therefore, same set of functions can be used for both types.

HITAG 1 and HITAG S transponder are available with different memory sizes. Due to this, the maximum address, which can be specified depends also on the specific type of transponder:

Type	Memory Size (Bits)	Memory Size (Bytes)	Valid Address Range
HITAG 1	2048	256	0-63
HITAG S 2048	2048	256	0-63
HITAG S 256	256	32	0-7

10.1 Read/Write Data

10.1.1 Hitag1S_ReadPage

Read one page (4 bytes) from the transponder.

```
bool Hitag1S_ReadPage(int PageAddress,byte *Page);
```

Parameters:	
`int PageAddress`	Specifies the address of the page to be read.
`byte *Page`	Pointer to an array of 4 bytes where page data is stored after a successful operation.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

10.1.2 Hitag1S_WritePage

Write one page (4 bytes) to the transponder.

```
bool Hitag1S_WritePage(int PageAddress,const byte *Page);
```

Parameters:	
`int PageAddress`	Specifies the address of the page to be written.
`byte *Page`	Pointer to an array of 4 bytes which are written to the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

10.1.3 Hitag1S_ReadBlock

Read 1 to 4 consecutive pages (4 to 16 bytes) from the transponder. The number of pages depends on the specified address: The read process is stopped as soon as the read address reaches a block boundary, which is a multiple of 4. If `BlockAddress` already specifies a block boundary, the maximum of 4 pages will be read.

```
bool Hitag1S_ReadBlock(int BlockAddress,
                       byte *Block,int *BytesRead);
```

Parameters:	
`int BlockAddress`	Specifies the first page address of the block to be read.
`byte *Page`	Pointer to an array of 4 to 16 bytes which are read from the transponder.
`int *BytesRead`	Pointer to an integer, which receives the number of actually read bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

10.1.4 Hitag1S_WriteBlock

Write 1 to 4 consecutive pages (4 to 16 bytes) to the transponder. The number of pages depends on the specified address: The write process is stopped as soon as the write address reaches a block boundary, which is a multiple of 4. If `BlockAddress` already specifies a block boundary, the maximum of 4 pages will be written.

```
bool Hitag1S_WriteBlock(int BlockAddress,const byte *Block,
                        int *BytesWritten);
```

Parameters:	
`int BlockAddress`	Specifies the first page address of the block to be written.
`byte *Page`	Pointer to an array of 4 to 16 bytes which are written to the transponder.
`int *BytesWritten`	Pointer to an integer, which receives the number of actually written bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

10.2 Hitag1S_Halt

This functions will halt a currently selected transponder. The transponder will not participate in any further transponder communication till the RF field is turned off and on again.

```
bool Hitag1S_Halt(void);
```

Parameters:	None.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

11 HITAG 2-Specific Transponder Operations

This chapter describes functions for accessing HITAG 2 transponders.

HITAG 2 is a transponder with a memory size of 256 bits, thus 32 bytes. It stores data organized in pages, where one page is 4 bytes. There are 8 pages, which can be accessed with addresses in the range from 0 to 7.

HITAG 2 can be operated in two modes: Password mode and crypto mode.

Note:

TWN4 supports password mode of HITAG 2 only.

11.1 Read/Write Data

11.1.1 Hitag2_ReadPage

Read one page (4 bytes) from the transponder.

```
bool Hitag2_ReadPage(int PageAddress,byte *Page);
```

Parameters:	
`byte PageAddress`	Specifies the address of the page to be read.
`byte *Page`	Pointer to an array of 4 bytes where page data is stored after a successful operation.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

11.1.2 Hitag2_WritePage

Write one page (4 bytes) to the transponder.

```
bool Hitag2_WritePage(byte PageAddress,const byte *Page);
```

Parameters:	
`byte PageAddress`	Specifies the address of the page to be written.
`byte *Page`	Pointer to an array of 4 bytes which are written to the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

11.1.3 Hitag2_SetPassword

During search for HITAG 2, TWN4 is using a password for doing a login to the transponder. The default password after a reset is `0x4D,0x49,0x4B,0x52`. This is the well-known default password for HITAG 2.

```
void Hitag2_SetPassword(const byte *Password);
```

Parameters:	
`const byte *Password`	Pointer to an array of 4 bytes, which contains the new password.
Return:	None.

11.2 Hitag2_Halt

This functions will halt a currently selected transponder. The transponder will not participate in any further transponder communication till the RF field is turned off and on again.

```
bool Hitag2_Halt(void);
```

Parameters:	None.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

12 EM4x50-Specific Transponder Operations

This chapter describes functions for accessing EM4x50 transponders. There are several chips, which are compatible to each other within this family. These are: EM4050, EM4150, EM4450, EM4550. According to the datasheet of the EM4x50 transponder, one word is meant to be 4 bytes.

12.1 Functions

Perform a login operation to the transponder.

12.1.1 EM4150_Login

```
bool EM4150_Login(const byte *Password)
```

Parameters:	
`const byte *Password`	Pointer to an array of 4 bytes which contains the password.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

12.1.2 EM4150_ReadWord

Read one word (4 bytes) from the transponder.

```
bool EM4150_ReadWord(int Address,byte *Word)
```

Parameters:	
`int Address`	Specifies the address of the page to be read. The valid address range is from 0 to 33.
`byte *Word`	Pointer to an array of 4 bytes which contains data read from the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

12.1.3 EM4150_WriteWord

Write one word (4 bytes) to the transponder.

```
bool EM4150_WriteWord(int Address,const byte *Word)
```

Parameters:	
`int Address`	Specifies the address of the page to be written.
`const byte *WordData`	Pointer to an array of 4 bytes which contains data to be written to the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

12.1.4 EM4150_WritePassword

Change the password stored on a transponder.

```
bool EM4150_WritePassword(const byte *ActualPassword,const byte *NewPassword)
```

Parameters:	
`const byte *ActualPassword`	Pointer to an array of 4 bytes which specifies the current password of the transponder.
`const byte *NewPassword`	Pointer to an array of 4 bytes which specifies the password to be written to the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

12.1.5 EM4150_GetTagInfo

Get detailed information regarding transponder type `LFTAG_EM4150` being found by function `SearchTag`.

```
int EM4150_GetTagInfo(void)
```

Parameters:	None.
Return:	One of the following pre-defined values: `EM4150_BITRATE_UNKNOWN` or `EM4150_BITRATE_F64` or `EM4150_BITRATE_F40`.

13 AT55xx-Specific Transponder Operations

This chapter describes functions for accessing AT55xx transponders. There are several chips, which are compatible to each other within this family. These are: e5550, e5551, T5555, T5555B, T5556, T5557, ATA5567, ATA5577. Note: T5552 and T5558 are not supported by this API.

13.1 Control Functions

13.1.1 AT55_Begin

The function `AT55_Begin` must be used before subsequent read or write access to the transponder in question.

```
void AT55_Begin(void);
```

Parameters:	None.
Return:	None.

Background:

Normally, in order to begin any read/write access to a transponder, the TWN4 system provides the function `SearchTag`. This function searches for a transponder and keeps the RF in appropriate condition to allow subsequent read- and write access.

This sequence is not applicable for the AT55xx family of transponders for two reasons:

- The transponder does not contain a serial number
- The transponder does not send data in a well-known standard format

The way out of this situation is the function `AT55_Begin`, which does not return any transponder data but turns on RF field for subsequent read-/write operations.

13.2 Read Data

Requirements:

The firmware of TWN4 supports read of data only, if the modulation of the transponder is configured to manchester coding with a bitrate of RF/128 up to RF/8.

Furthermore, TWN4 can be set up to support sequence terminator turned on or off.

The default setup is RF/64 with sequence terminator turned off. In order to choose a different configuration the function `SetParameters` must be used. Here is an example of how use of RF/32 is programmed:

```
const byte MyRF32Config[] = { AT55_BITRATE, 1, 32, TLV_END };
SetParameters(MyRF32Config,sizeof(MyRF32Config));
```

Example of how to turn on sequence terminator on and use RF/40:

```
const byte MyRF40Config[] =
        {
                AT55_OPTIONS, 1, AT55_OPT_SEQUENCETERMINATOR_ON,
                AT55_BITRATE, 1, 40,
                TLV_END
        };
SetParameters(MyRF40Config,sizeof(MyRF40Config));
```

13.2.1 AT55_ReadBlock

Read one block (4 bytes) from the transponder.

```
bool AT55_ReadBlock(int Address,byte *Data);
```

Parameters:	
`int Address`	Specifies the address of the page to be read.
`byte *Data`	Pointer to an array of 4 bytes which contains data read from the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

13.2.2 AT55_ReadBlockProtected

Read one block (4 bytes) from a password-protected transponder.

```
bool AT55_ReadBlockProtected(int Address,byte *Data,const byte *Password);
```

Parameters:	
`int Address`	Specifies the address of the page to be read.
`byte *Data`	Pointer to an array of 4 bytes which contains data read from the transponder.
`const byte *Password`	Pointer to an array of 4 bytes which contains the password.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

13.3 Write Data

13.3.1 AT55_WriteBlock

Write one block (4 bytes) to the transponder.

```
bool AT55_WriteBlock(int Address,const byte *Data);
```

Parameters:	
`int Address`	Specifies the address of the page to be written.
`const byte *Data`	Pointer to an array of 4 bytes which contains data to be written to the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

13.3.2 AT55_WriteBlockProtected

Write one block (4 bytes) to a password-protected transponder.

```
bool AT55_WriteBlockProtected(int Address,const byte *Data,const byte *Password);
```

Parameters:	
`int Address`	Specifies the address of the page to be written.
`const byte *Data`	Pointer to an array of 4 bytes which contains data to be written to the transponder.
`const byte *Password`	Pointer to an array of 4 bytes which contains the password.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

13.3.3 AT55_WriteBlockAndLock

Write one block (4 bytes) to a transponder and lock the written data. Locking data means, that it is not possible to modify data contained in this block.

```
bool AT55_WriteBlockAndLock(int Address,const byte *Data);
```

Parameters:	
`int Address`	Specifies the address of the page to be written.
`const byte *Data`	Pointer to an array of 4 bytes which contains data to be written to the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

13.3.4 AT55_WriteBlockProtectedAndLock

Write one block (4 bytes) to a password-protected transponder and lock the written data. Locking data means, that it is not possible to modify data contained in this block.

```
bool AT55_WriteBlockProtectedAndLock(int Address,const byte *Data,const byte *Password);
```

Parameters:	
`int Address`	Specifies the address of the page to be written.
`const byte *Data`	Pointer to an array of 4 bytes which contains data to be written to the transponder.
`const byte *Password`	Pointer to an array of 4 bytes which contains the password.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14 TILF (TIRIS) Functions

This chapter describes functions for accessing Texas Instruments Low Frequency transponders (TILF). This type of transponder was formerly also known as TIRIS.

Note:

It is highly recommended to also study datasheets of according transponders. Datasheets are available from Texas Instruments.

14.1 Search Function

14.1.1 TILF_SearchTag

Search for a TILF tag. This function can be used directly instead of the general search function `SearchTag`. The function doing a search for a TILF tag in two different ways: First, a tag is search via a call of function `TILF_ChargeOnlyRead`. Second, a tag is searched via function `TILF_MUGeneralReadPage`, address 3.

```
bool TILF_SearchTag(int *IDBitCount,byte *ID,int MaxIDBytes);
```

Parameters:	
`int *IDBitCount`	A pointer to an integer, which receives the number of actually read bits(!). Due to the nature of the functions `TILF_ChargeOnlyRead` and `TILF_MUGeneralReadPage`, the number of received bits is either 32 or 64.
`byte *ID`	A pointer to an array of bytes, which receives the read ID. Due to the nature of the functions `TILF_ChargeOnlyRead` and `TILF_MUGeneralReadPage`, the number of received bytes is either 4 or 8.
`int MaxIDBytes`	The maximum number of bytes, which will be copied to `ID`
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.2 Single-Page Read/Write Function

14.2.1 TILF_ChargeOnlyRead

Search for a single page transponder. This might be a read-only or a read/write transponder. Only transponders are detected, where ID is stored under use of a CCITT CRC. If a transponder is programmed in a different way, consider using `TILF_ChargeOnlyReadLo`, which allows to read entire content of transponder W/O CRC check.

```
bool TILF_ChargeOnlyRead(byte *ReadData);
```

Parameters:	
`byte *ReadData`	A pointer to an array of 8 bytes, which receives checked ID data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.2.2 TILF_ChargeOnlyReadLo

Search for a single page transponder. This might be a read-only or a read/write transponder. No CRC check is performed, thus allowing to read also custom programmed tags. The interpretation of data should be known by the solution builder.

```
bool TILF_ChargeOnlyReadLo(byte *ReadData);
```

Parameters:	
`byte *ReadData`	A pointer to an array of 16 bytes, which receives unchecked ID data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.2.3 TILF_SPProgramPage

Write data to a single-page read/write transponder by using CCITT CRC.

```
bool TILF_SPProgramPage(const byte *WriteData,byte *ReadData);
```

Parameters:	
`const byte *WriteData`	A pointer to an array of 8 bytes, which will be written to the transponder.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives checked response from the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.2.4 TILF_SPProgramPageLo

Write data to a single-page read/write transponder.

```
bool TILF_SPProgramPageLo(const byte *WriteData,byte *ReadData);
```

Parameters:	
`const byte *WriteData`	A pointer to an array of 10 bytes, which will be written to the transponder.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives unchecked response from the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3 Multi-Page Read/Write Function

14.3.1 TILF_MPGeneralReadPage

General read of data from a multi-page transponder (MPT).

```
bool TILF_MPGeneralReadPage(int Address,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.2 TILF_MPSelectiveReadPage

Selective read of data from a multi-page transponder (SAMPT or SAMPTS).

```
bool TILF_MPSelectiveReadPage(
                int Address,const byte *SelectiveAddress,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`const byte *SelectiveAddress`	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.3 TILF_MPProgramPage

Program one page to a multi-page transponder (MPT).

```
bool TILF_MPProgramPage(
                int Address,const byte *WriteData,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`const byte *WriteData`	A pointer to an array of 8 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.4 TILF_MPSelectiveProgramPage

Selective program of one page to a multi-page transponder (SAMPT or SAMPTS).

```
bool TILF_MPSelectiveProgramPage(
                int Address,const byte *SelectiveAddress,
                const byte *WriteData,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`const byte *SelectiveAddress`	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
`const byte *WriteData`	A pointer to an array of 8 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.5 TILF_MPLockPage

Lock one page on a multi-page transponder (MPT).

```
bool TILF_MPLockPage(int Address,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.6 TILF_MPSelectiveLockPage

Selective lock one page on a multi-page transponder (SAMPT or SAMPTS).

```
bool TILF_MPSelectiveLockPage(
                int Address,const byte *SelectiveAddress,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`const byte *SelectiveAddress`	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
`byte *ReadData`	A pointer to an array of 8 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.7 TILF_MPGeneralReadPageLo

General read of data from a multi-page transponder (MPT) W/O CRC-check.

```
bool TILF_MPGeneralReadPageLo(int Address,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.8 TILF_MPSelectiveReadPageLo

Selective read of data from a multi-page transponder (SAMPT or SAMPTS) W/O CRC-check.

```
bool TILF_MPSelectiveReadPageLo(
                int Address,const byte *SelectiveAddress,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`const byte *SelectiveAddress`	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.9 TILF_MPProgramPageLo

Program one page to a multi-page transponder (MPT) W/O CRC-check.

```
bool TILF_MPProgramPageLo(
                int Address,const byte *WriteData,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`const byte *WriteData`	A pointer to an array of 10 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.10 TILF_MPSelectiveProgramPageLo

Selective program of one page to a multi-page transponder (SAMPT or SAMPTS) W/O CRC-check.

```
bool TILF_MPSelectiveProgramPageLo(
                int Address,const byte *SelectiveAddress,
                const byte *WriteData,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`const byte *SelectiveAddress`	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
`const byte *WriteData`	A pointer to an array of 10 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.11 TILF_MPLockPageLo

Lock one page on a multi-page transponder (MPT) W/O CRC-check.

```
bool TILF_MPLockPageLo(int Address,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.3.12 TILF_MPSelectiveLockPageLo

Selective lock one page on a multi-page transponder (SAMPT or SAMPTS) W/O CRC-check.

```
bool TILF_MPSelectiveLockPageLo(
                int Address,const byte *SelectiveAddress,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`const byte *SelectiveAddress`	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
`byte *ReadData`	A pointer to an array of 16 bytes, which receives data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4 Multi-Usage Read/Write Function

14.4.1 TILF_MUGeneralReadPage

General read of one page from a multi-usage transponder (MUSA).

```
bool TILF_MUGeneralReadPage(int Address,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.2 TILF_MUSelectiveReadPage

Selective read of one page from a multi-usage transponder (MUSA).

```
bool TILF_MUSelectiveReadPage(
                int Address,int SelectiveAddress,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`int SelectiveAddress`	A value which specifies the 8-bit selective address.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.3 TILF_MUSpecialReadPage

Special read of one page from a multi-usage transponder (MUSA).

```
bool TILF_MUSpecialReadPage(
                int Address,const byte *SpecialAddress1,
                const byte *SpecialAddress2,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be read from.
`const byte *SpecialAddress1`	Pointer to an array of 5 bytes (40 bits) which provides the special address 1.
`const byte *SpecialAddress2`	Pointer to an array of 3 bytes (24 bits) which provides the special address 2.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.4 TILF_MUProgramPage

Program one page to a multi-usage transponder (MUSA).

```
bool TILF_MUProgramPage(int Address,const byte *WriteData,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`const byte *WriteData`	A pointer to an array of 5 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.5 TILF_MUSelectiveProgramPage

Selective program of one page to a multi-usage transponder (MUSA).

```
bool TILF_MUSelectiveProgramPage(
                int Address,int SelectiveAddress,
                const byte *WriteData,byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`int SelectiveAddress`	A value which specifies the 8-bit selective address.
`const byte *WriteData`	A pointer to an array of 5 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.6 TILF_MUSpecialProgramPage

Special program of one page to a multi-usage transponder (MUSA).

```
bool TILF_MUSpecialProgramPage(
                int Address,const byte *SpecialAddress1,
                const byte *SpecialAddress2,const byte *WriteData,
                byte *ReadData);
```

Parameters:	
`int Address`	The page address, where data will be programmed to.
`const byte *SpecialAddress1`	Pointer to an array of 5 bytes (40 bits) which provides the special address 1.
`const byte *SpecialAddress2`	Pointer to an array of 3 bytes (24 bits) which provides the special address 2.
`const byte *WriteData`	A pointer to an array of 5 bytes, which will be programmed.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.7 TILF_MULockPage

Lock one page of a multi-usage transponder (MUSA).

```
bool TILF_MULockPage(int Address,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.8 TILF_MUSelectiveLockPage

Selective lock of one page of a multi-usage transponder (MUSA).

```
bool TILF_MUSelectiveLockPage(
                int Address,int SelectiveAddress,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`int SelectiveAddress`	A value which specifies the 8-bit selective address.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

14.4.9 TILF_MUSpecialLockPage

Special lock of one page of a multi-usage transponder (MUSA).

```
bool TILF_MUSpecialLockPage(
                int Address,const byte *SpecialAddress1,
                const byte *SpecialAddress2,byte *ReadData);
```

Parameters:	
`int Address`	The page address, which will be locked.
`const byte *SpecialAddress1`	Pointer to an array of 5 bytes (40 bits) which provides the special address 1.
`const byte *SpecialAddress2`	Pointer to an array of 3 bytes (24 bits) which provides the special address 2.
`byte *ReadData`	A pointer to an array of 7 bytes, which receives page data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15 ISO14443 Transponder Operations

This chapter handles specific operations for transparent access of ISO14443A/B compliant transponders. Before these functions can be used, the transponder must have been selected using the function `SearchTag(...)`.

15.1 ISO14443A

15.1.1 Get ATQA

This function delivers the ATQA (Answer To Request TypeA) of the last detected ISO14443A compliant transponder.

```
bool ISO14443A_GetATQA(byte* ATQA);
```

Parameters:	
`byte* ATQA`	After successful completion of this function, the buffer referred by this parameter holds the ATQA of the transponder. The function returns two bytes of data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.1.2 Get SAK

This function delivers the SAK (Select Acknowledge) of the last detected ISO14443A compliant transponder.

```
bool ISO14443A_GetSAK(byte* SAK);
```

Parameters:	
`byte* SAK`	After successful completion of this function, the buffer referred by this parameter holds the SAK of the transponder. The function returns one byte of data.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.1.3 Get ATS

This function delivers the ATS (Answer To Select) of a ISO14443A layer 4 transponder.

```
bool ISO14443A_GetATS
    (
    byte* ATS,
    int* ATSByteCnt,
    int MaxATSByteCnt
    );
```

Parameters:

`byte* ATS`	After successful completion of this function, the buffer referred by this parameter holds the ATS which was read from the transponder. Take care for adequate dimensioning.
`int* ATSByteCnt`	After successful completion of this function, this parameter holds the number of bytes, the ATS contains.
`int MaxATSByteCnt`	This parameter holds the array-size of `ATS` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.2 ISO14443B

15.2.1 Get ATQB

This function delivers the ATQB (Answer To Request TypeB) of the last detected ISO14443B compliant transponder.
Note: This function cannot be called on TWN4 MultiTech Legic.

```
bool ISO14443B_GetATQB(byte* ATQB, int* ATQBByteCnt, int MaxATQBByteCnt);
```

Parameters:

`byte* ATQB`	After successful completion of this function, the buffer referred by this parameter holds the ATQB of the transponder. Take care for adequate dimensioning, the ATQB usually has 12 or 13 bytes in length.
`int* ATQBByteCnt`	After successful completion of this function, this parameter holds the number of bytes of `ATQB`.
`int MaxATQBByteCnt`	This parameter holds the array-size of `ATQB` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.2.2 Get Answer to ATTRIB

This function delivers the transponder's answer to the ATTRIB command, which is sent automatically during selection process by the reader.
Note: This function cannot be called on TWN4 MultiTech Legic.

```
bool ISO14443B_GetAnswerToATTRIB
    (
    byte* AnswerToATTRIB,
    int* AnswerToATTRIBByteCnt,
    int MaxAnswerToATTRIBByteCnt
    );
```

Parameters:	
`byte* AnswerToATTRIB`	After successful completion of this function, the buffer referred by this parameter holds the `AnswerToATTRIB` of the transponder. Take care for adequate dimensioning, `AnswerToATTRIB` can have one or more bytes in length.
`int* AnswerToATTRIBByteCnt`	After successful completion of this function, this parameter holds the number of bytes of `AnswerToATTRIB`.
`int MaxAnswerToATTRIBByteCnt`	This parameter holds the array-size of `AnswerToATTRIB` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.3 Check Presence

This function can be used to probe if a ISO14443-4 transponder is still in reading range. The internal state of the transponder remains unchanged.
Note: This function cannot be called on TWN4 MultiTech Legic.

```
bool ISO14443_4_CheckPresence(void);
```

Parameters:	None.
Return:	If the transponder is still in range, the return value is `true`, otherwise it is `false`.

15.4 ISO14443-3 Transparent Data Exchange

This function can be used for transparent exchange of data between reader and ISO14443-3 transponders. The function does not calculate any CRC or other overhead by itself, so if necessary this has to be conducted on host side.

```
bool ISO14443_3_TDX
    (
    byte* TXRX,
    int TXByteCnt,
    int* RXByteCnt,
    int MaxRXByteCnt
    );
```

Parameters:

`byte* TXRX`	This buffer holds the byte-string that shall be transmitted to the transponder. The response of the transponder is also returned by this parameter. Take care for adequate dimensioning.
`int TXByteCnt`	This parameter holds the number of bytes that shall be transmitted to the transponder.
`int* RXByteCnt`	After successful completion of this function, this parameter holds the number of bytes that the transponder response contains.
`int MaxRXByteCnt`	This parameter holds the array-size of `TXRX` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.5 ISO14443-4 Transparent Data Exchange

This function can be used for transparent exchange of data between reader and ISO14443-4 transponders. All framing of layer 4 subset is already done by the reader, so only the payload needs to be passed to the function.

```
bool ISO14443_4_TDX
    (
    byte* TXRX,
    int TXByteCnt,
    int* RXByteCnt,
    int MaxRXByteCnt
    );
```

Parameters:

`byte* TXRX`	This buffer holds the byte-string that shall be transmitted to the transponder. The response of the transponder is also returned by this parameter. Take care for adequate dimensioning.
`int TXByteCnt`	This parameter holds the number of bytes that shall be transmitted to the transponder.
`int* RXByteCnt`	After successful completion of this function, this parameter holds the number of bytes that the transponder response contains.
`int MaxRXByteCnt`	This parameter holds the array-size of `TXRX` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.6 Multiple Tag Handling

TWN4 is capable of handling multiple ISO14443A tags that are simultaneously present in the RF field. Use the following functions to inventorize the field and select one of the discovered transponders for subsequent operations.

15.6.1 Search for Transponders

Use this function to search the RF field for ISO14443A transponders. The result is a list of the UID of the respective transponders.

```
bool ISO14443A_SearchMultiTag
    (
    int* UIDCnt,
    int* UIDListByteCnt,
    byte* UIDList,
    int MaxUIDListByteCnt
    );
```

Parameters:	
`int* UIDCnt`	This parameter holds the number of found transponder UIDs.
`int* UIDListByteCnt`	This parameter holds the number of valid bytes of the UID list.
`byte* UIDList`	This parameter holds the list of found UIDs. Every entry is preceeded by a single byte representing the respective UID length, e.g. the two transponder IDs 11223344 and 00010203040506 would be coded as follows: 041122334070001020304050 6.
`int MaxUIDListByteCnt`	This parameter holds the array-size of `UIDList` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

15.6.2 Select Transponder

Use this function to select one of the discovered transponders for further operations.

```
bool ISO14443A_SelectTag(const byte* UID, int UIDByteCnt);
```

Parameters:	
`const byte* UID`	Specify the UID of the transponder to be selected.
`int UIDByteCnt`	This parameter holds the byte count of the specified UID.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

16 MIFARE Classic Specific Transponder Operations

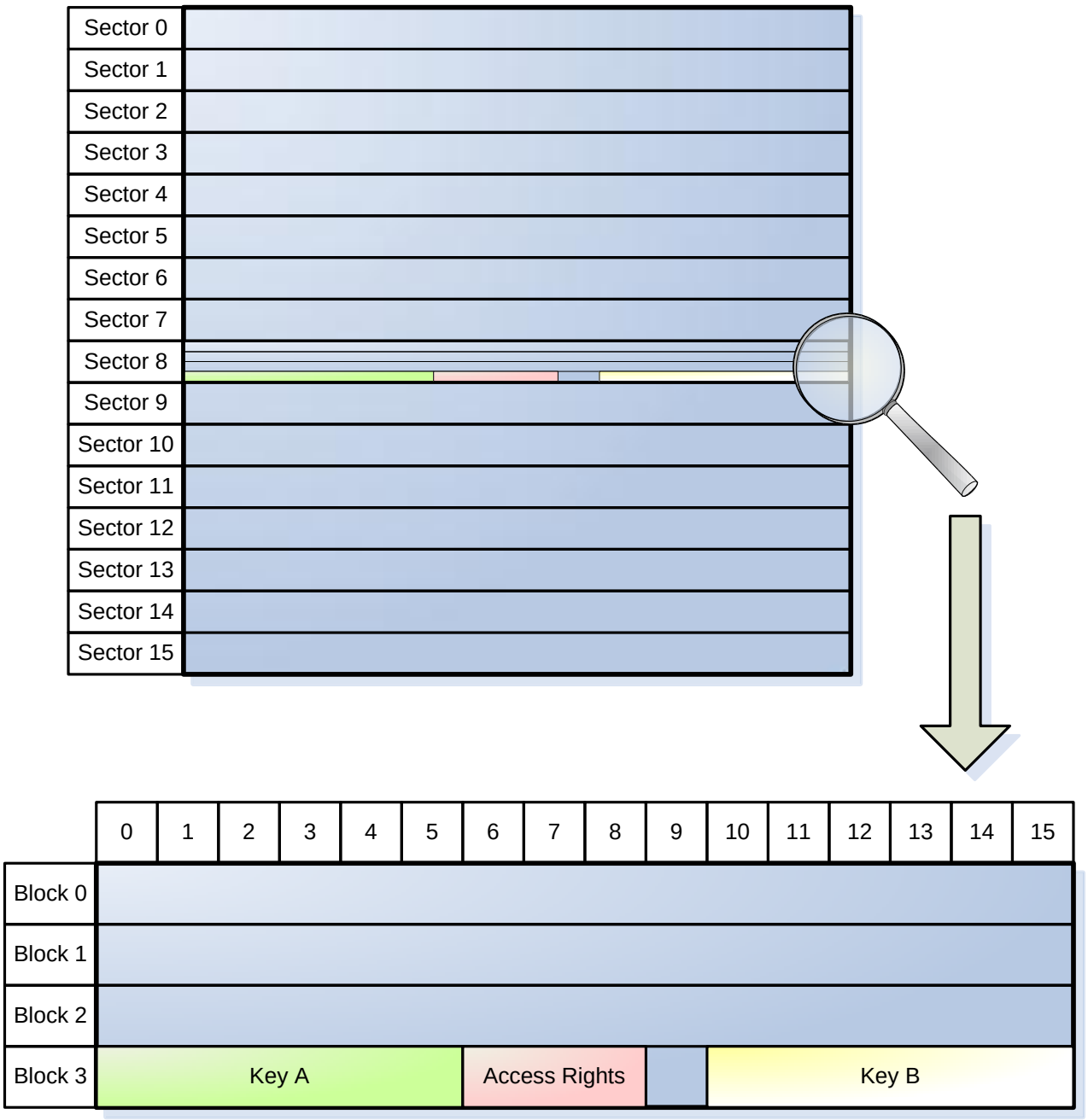


Figure 16.1: Memory layout of a MIFARE Classic 1K transponder

The memory of MIFARE Classic transponders is organized in sectors and blocks. In case of MIFARE Classic 1K, the memory is divided into 16 sectors, each sector holds 4 blocks. Each block holds 16 bytes of data. Each sector is secured by two keys, Key A and Key B which are always located in the last block of a sector (sector trailer). In order to access the respective sector, a login using one of the two keys has to be performed. Once logged in, the data blocks are accessible for read-, write- or value-operations. Each key may be equipped with certain access rights, the access rights are coded in byte 6, 7 and 8 of the sector trailer. Byte 9 is available for data storage.
In case of MIFARE Classic 4K, the memory layout of sector addresses 0 to 31 is compatible to the 1K version, from sector 32 to 39, each sector holds 16 data blocks.
In any case, block 0 of sector 0 is called manufacturer block, and cannot be overwritten. Within this block, the UID is stored and some manufacturer specific data.

16.1 Login

In order to do any operation on a sector of a MIFARE Classic transponder, a login to the respective sector has to be performed. Each sector holds two keys, *Key A* and *Key B*. Depending on the access conditions of the sector, the appropriate key shall be used for the desired operation. Both the keys and the access conditions are stored in the sector trailer.

```
bool MifareClassic_Login
    (
    const byte* Key,
    byte KeyType,
    int Sector
    );
```

Parameters:

`const byte* Key`	Pointer to an array of bytes, which has to contain six bytes. These bytes represent the key for the login process.
`byte KeyType`	Specifies, with which key the operation has to be performed. This is one of the defined constants `KEYA` or `KEYB`.
`int Sector`	Specifies the sector for the login.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Key (hex)	Description
FF FF FF FF FF FF	Default Transport Key A/B (NXP)
A0 A1 A2 A3 A4 A5	Default Transport Key A (Infineon)
B0 B1 B2 B3 B4 B5	Default Transport Key B (Infineon)
D3 F7 D3 F7 D3 F7	Default key for NDEF-formatted tags

Table 16.1: Well-known keys for MIFARE Classic transponders

16.2 Read/Write Data

16.2.1 Read Data Block

Read 16 bytes of data from a data-block of the transponder. Please note: If a sector trailer is read, the respective key which was used for login is represented by zeros.

```
bool MifareClassic_ReadBlock
    (
    int Block,
    byte* Data
    );
```

Parameters:

`int Block`	Specify the address of the block to be read. The valid range of this parameter is between 0 and 255.
`byte* Data`	This parameter holds the data which was read from the tag if the operation was successful. Note that this function always reads 16 bytes of data, so the minimum array size of `Data` must be at least 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

16.2.2 Write Data Block

Write 16 bytes of data to a data-block of the transponder. Special care must be taken when writing to a sector trailer as a faulty setting of the access conditions can make the sector unaccessible.

```
bool MifareClassic_WriteBlock
    (
    int Block,
    const byte* Data
    );
```

Parameters:

`int Block`	Specify the address of the block to be written. The valid range of this parameter is between 0 and 255.
`const byte* Data`	This parameter holds the data which shall be written to the tag. Note that this function always writes 16 bytes of data, so the minimum array size of `Data` shall be at least 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

16.3 Handling of Value Blocks

16.3.1 Read Value Block

Read the value stored in a MIFARE Classic compliant value block.

```
bool MifareClassic_ReadValueBlock
    (
    int Block,
    int* Value
    );
```

Parameters:

`int Block`	Specify the address of the block to be read. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int* Value`	This parameter holds the value which was read from the tag if the operation was successful.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: This function checks if the block has a valid value block format. If this is not the case, the function returns `false`.

16.3.2 Write Value Block

Format a data block to a MIFARE Classic compliant value block and assign an initial value.

```
bool MifareClassic_WriteValueBlock
    (
    int Block,
    int Value
    );
```

Parameters:

`int Block`	Specify the address of the block to be formatted. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int Value`	This parameter holds the initial value of the value block.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

16.3.3 Increment Value Block

Credit a value block with a given increment value.

```
bool MifareClassic_IncrementValueBlock
    (
    int Block,
    int Value
    );
```

Parameters:

`int Block`	Specify the address of the block to be incremented. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int Value`	This parameter holds the increment value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

16.3.4 Decrement Value Block

Debit a value block with a given decrement value.

```
bool MifareClassic_DecrementValueBlock
    (
    int Block,
    int Value
    );
```

Parameters:

`int Block`	Specify the address of the block to be decremented. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int Value`	This parameter holds the decrement value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

16.3.5 Copy Value Block

Copy a value block within a sector.

```
bool MifareClassic_CopyValueBlock
    (
    int SourceBlock,
    int DestBlock
    );
```

Parameters:

`int SourceBlock`	Specify the address of the source block.
`int DestBlock`	Specify the address of the destination block.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17 MIFARE Plus Specific Transponder Operations

MIFARE Plus is mostly compatible to MIFARE Classic, but comes with several enhancements regarding security and functionality. The memory layout is compatible to MIFARE Classic. MIFARE Plus transponders incorporate four different levels of operation, these are called Security Level (SL).
Blank transponders are usually sold in SL0, which is used for personalisation of the transponder. Within this level, the keys and data blocks can be written. When the personalisation process has finished, the transponder has to be switched to a higher security level.
In usual cases, this is SL1 where the transponder is compatible to Mifare Classic, this means the login process, memory layout and read/write operations are the same. In this case refer to the API description of MIFARE Classic.
In case of MIFARE Plus X, the transponder may be switched from SL1 to SL2 where an additional AES authentication becomes necessary before any memory operation is possible. All subsequent Crypto1 operations are then depending on this authentication, as a session key is calculated and the Crypto1 key is diversified for this session. So, after AES authentication, the API functions for MIFARE Classic have to be used for accessing the memory.
MIFARE Plus S or X can be switched to SL3, where a AES authentication is necessary to access the transponder memory. In case of MIFARE Plus X all operations are done fully encrypted, in case of MIFARE Plus S all operations are done in plain but with computation of an additional MAC. For memory operations in SL3, the API functions described in the following chapters shall be used.
Please note, once a MIFARE Plus transponder has been switched to a higher security level, it cannot be switched back again.

17.1 Personalisation

Personalisation can only be done if the transponder is operating in SL0. As all communication is done in plain, this process should be conducted at a secure place. When all personalisation data has been written, the personalisation must be finished by issuing the function Commit Personalisation. After that, the personalisation becomes valid and the transponder is switched to SL1.

17.1.1 Write Personalisation

Use this function to write any personalisation data to a specific block of the transponder.

```
bool MFP_WritePerso(int BlockNr, const byte* Data);
```

Parameters:	
`int BlockNr`	Specify the block number to be written. This can either be the number of a sector block or a AES key.
`const byte* Data`	Specify the data to be written with this parameter. The function expects always 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.1.2 Commit Personalisation

This function shall be used to switch the transponder to SL1 when all personalisation has been finished. After calling this function, the transponder has to be reselected in order to access it again.

```
bool MFP_CommitPerso(void);
```

Parameters:	None
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.2 Authenticate AES

Use this function to do a mutual authentication in AES with the transponder. The key may either be a sector key or a special one like a level switch key. In case of MIFARE Plus running in SL2, a preceding AES authentication is necessary before any following memory operations which are conducted in Crypto1. A typical transaction flow looks like this:

Search Tag
⇓
Authenticate AES
⇓
Mifare Classic Login
⇓
Mifare Classic Read/Write Data

```
bool MFP_Authenticate(int CryptoEnv, int KeyBNr, const byte* Key);
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. All consecutive operations with the transponder shall be done using the specified environment.
`int KeyBNr`	Specify the key number that shall be used for authentication. This can either be a sector key or a special key like a level switch key.
`const byte* Key`	Specify the key that shall be used for authentication. For AES, the key must have a key length of 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.3 Security Level 3

In Security Level 3 all memory related operations require a preceding AES authentication with the respective key. Please note, MIFARE Plus S does not support all the functionality of a MIFARE Plus X, e.g. handling of value blocks is not supported here.

17.3.1 Read/Write Data

17.3.1.1 Read Data Block

Use this function to read a data block from a MIFARE Plus transponder.

```
bool MFP_ReadBlock(int CryptoEnv, int Block, byte* Data);
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int Block`	Specify the number of the block that shall be read.
`byte* Data`	This parameter holds the data which was read from the tag if the operation was successful. Note that this function always reads 16 bytes of data, so the minimum array size of `Data` must be at least 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.3.1.2 Write Data Block

Use this function to write data to a block of a MIFARE Plus transponder.

```
bool MFP_WriteBlock(int CryptoEnv, int Block, const byte* Data);
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int Block`	Specify the number of the block that shall be written.
`const byte* Data`	This parameter holds the data which shall be written to the tag. Note that this function always writes 16 bytes of data, so the minimum array size of `Data` must be at least 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.3.2 Handling of Value Blocks

17.3.2.1 Read Value Block

Use this function to read the value stored in a MIFARE compliant value block.

```
bool MFP_ReadValueBlock(int CryptoEnv, int Block, int* Value);
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int Block`	Specify the address of the block to be read. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int* Value`	This parameter holds the value which was read from the tag if the operation was successful.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: This function checks if the block has a valid value block format. If this is not the case, the function returns `false`.

17.3.2.2 Write Value Block

Format a data block to a MIFARE compliant value block and assign an initial value.

```
bool MFP_WriteValueBlock(int CryptoEnv, int Block, int Value);
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int Block`	Specify the address of the block to be formatted. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int Value`	This parameter holds the initial value of the value block.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.3.2.3 Increment Value Block

Credit a value block with a given increment value.

```
bool MFP_IncrementValueBlock(int CryptoEnv, int Block, int Value);
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int Block`	Specify the address of the block to be incremented. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int Value`	This parameter holds the increment value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.3.2.4 Decrement Value Block

Debit a value block with a given decrement value.

```
bool MFP_DecrementValueBlock(int CryptoEnv, int Block, int Value);
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int Block`	Specify the address of the block to be decremented. The valid range of this parameter is between 0 and 255. Note that this function does not work with sector trailers.
`int Value`	This parameter holds the decrement value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

17.3.2.5 Copy Value Block

Copy a value block within a sector.

```
bool MFP_CopyValueBlock(int CryptoEnv, int SourceBlock, int DestBlock);
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int SourceBlock`	Specify the address of the source block.
`int DestBlock`	Specify the address of the destination block.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18 MIFARE Ultralight/Ultralight C/Ultralight EV1 Specific Transponder Operations

18.1 Authentication (Ultralight C only)

Depending on the security settings of the transponder, a login with the valid transponder key might be necessary prior performing any further operation.

18.1.1 Authentication with given Key

Use this function to authenticate at a Mifare Ultralight C transponder with a given key.

```
bool MifareUltralightC_Authenticate(const byte* Key);
```

Parameters:

`const byte* Key`	Pointer to an array of bytes, which has to contain 16 bytes. These bytes represent the key for the authentication process.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Key (hex)	Description
`49 45 4D 4B 41 45 52 42 21 4E 41 43 55 4F 59 46`	Default Transport Key

Table 18.1: Well-known key for MIFARE Ultralight C transponders

18.1.2 Authentication using SAM Card

Use this function to authenticate at a Mifare Ultralight C transponder with a key stored on a SAM card. Depending on the security settings of the SAM card, an additional authentication between reader an SAM might be necessary prior issuing this command.

```
bool MifareUltralightC_SAMAuthenticate
    (
    int KeyNo,
    int KeyVersion,
    const byte* DIVInput,
    int DIVByteCnt
    );
```

Parameters:

`int KeyNo`	Specify the number of the SAM key entry that shall be used for authentication.
`int KeyVersion`	Specify the key version of the SAM key entry that shall be used for authentication.
`const byte* DIVInput`	Specify optional diversification input used for authentication.
`int DIVByteCnt`	Specify the number of bytes for diversification input. Valid values are 0 to 31.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.2 Write Key from SAM to Transponder Key Storage Area

Use this function to transfer a key from a SAM card to the key storage area of a Mifare Ultralight C transponder. Please note that the key stored on the SAM card must be dumpable. Depending on the security settings of the SAM card, an additional authentication between reader an SAM might be necessary prior issuing this command.

```
bool MifareUltralightC_WriteKeyFromSAM
    (
    int KeyNo,
    int KeyVersion,
    const byte* DIVInput,
    int DIVByteCnt
    );
```

Parameters:	
`int KeyNo`	Specify the number of the SAM key entry that shall be transfered.
`int KeyVersion`	Specify the key version of the SAM key entry.
`const byte* DIVInput`	Specify optional diversification input.
`int DIVByteCnt`	Specify the number of bytes for diversification input. Valid values are 0 to 31.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.3 Read/Write Data

18.3.1 Read Page

Though the page size of this transponder family is 4 bytes, the transponder always returns 16 bytes of data. This is achieved by reading four consecutive data pages, e.g. if page 4 is to be read, the transponder also returns the content of page 5, 6 and 7. The transponder incorporates an integrated roll-back mechanism if reading is done beyond the last physical available page address. E.g., in case of reading page 14 of MIFARE Ultralight this would result in reading page 14, 15, 0, 1.

```
bool MifareUltralight_ReadPage
    (
    int Page,
    byte* Data
    );
```

Parameters:	
`int Page`	Specify the address of the page to be read. The valid range of this parameter is between 0 and 15 (Ultralight) or 0 and 43 (Ultralight C).
`byte* Data`	This parameter holds the data which was read from the tag if the operation was successful. Note that this function always reads 16 bytes of data, so the minimum array size of `Data` must be at least 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.3.2 Write Page

Write 4 bytes of data to a data-page of the transponder. Compared to the read-function, this function processes only one page at once.

```
bool MifareUltralight_WritePage
    (
    int Page,
    const byte* Data
    );
```

Parameters:	
`int Page`	Specify the address of the page to be written. The valid range of this parameter is between 2 and 15 (Ultralight) or 2 and 47 (Ultralight C).
`const byte* Data`	This parameter holds the data which shall be written to the tag. Note that this function always writes 4 bytes of data, so the minimum array size of `Data` must be at least 4 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4 Mifare Ultralight EV1

18.4.1 Fast Read

The Fast Read function reads a number of pages beginning at a starting page from the transponder.

```
bool MifareUltralightEV1_FastRead(int StartPage, int NumberOfPages, byte* Data);
```

Parameters:	
`int StartPage`	Specify the address of the starting page.
`int NumberOfPages`	Specify the number of pages to be read.
`byte* Data`	This buffer holds the received data from the tag. Take care for proper dimensioning, the buffer size must be at least NumberOfPages * 4.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4.2 Increment Counter

Use this function to increment of the 3 one-way counters.

```
bool MifareUltralightEV1_IncCounter(int CounterAddr, int IncrValue);
```

Parameters:	
`int CounterAddr`	Specify the address of the counter to be incremented.
`int IncrValue`	Specify the increment value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4.3 Read Counter

Use this function to read the value of one of the 3 one-way counters.

```
bool MifareUltralightEV1_ReadCounter(int CounterAddr, int* CounterValue);
```

Parameters:	
`int CounterAddr`	Specify the address of the counter to be read.
`int* CounterValue`	This parameter holds the returned counter value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4.4 Read ECC Signature

Use this function to read the factory programmed 32 byte ECC signature, to verify NXP Semiconductors as the silicon vendor.

```
bool MifareUltralightEV1_ReadSig(byte* ECCSig);
```

Parameters:	
`byte* ECCSig`	This buffer holds the returned ECC signature. The required buffer size is 32 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4.5 Get Transponder Information

Use this function to retrieve information about the transponder such as product version or storage size.

```
bool MifareUltralightEV1_GetVersion(byte* Version);
```

Parameters:	
`byte* Version`	This buffer holds 8 bytes of version information.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4.6 Password Authentication

Use this function for password authentication. For authentication, a 4 bytes password and a 2 bytes acknowledge are required.

```
bool MifareUltralightEV1_PwdAuth(const byte* Password, const byte* PwdAck);
```

Parameters:	
`const byte* Password`	The 4 bytes password is specified by this parameter.
`const byte* PwdAck`	This buffer holds 2 bytes of Password Acknowledge.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

18.4.7 Check Tearing Event

Use this function to check if a tearing event has happened at a specific counter.

```
bool MifareUltralightEV1_CheckTearingEvent(int CounterAddr, byte* ValidFlag);
```

Parameters:	
`int CounterAddr`	Specify the address of the counter to be checked.
`byte* ValidFlag`	The `ValidFlag` is returned by this parameter. If no tearing event has happened, the returned value is 0xBD.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19 NTAG Specific Transponder Operations

19.1 Read/Write Data

19.1.1 Read Page

Though the page size of this transponder family is 4 bytes, the transponder always returns 16 bytes of data. This is achieved by reading four consecutive data pages, e.g. if page 4 is to be read, the transponder also returns the content of page 5, 6 and 7. The transponder incorporates an integrated roll-back mechanism if reading is done beyond the last physical available page address. The function is available for all members of the NTAG transponder family.

```
bool NTAG_ReadPage(int Page, byte* Data);
```

Parameters:	
`int Page`	Specify the address of the page to be read. The valid range of this parameter depends on the transponder type.
`byte* Data`	This parameter holds the data which was read from the tag if the operation was successful. Note that this function always reads 16 bytes of data, so the minimum array size of `Data` must be at least 16 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.1.2 Write Page

Write 4 bytes of data to a data-page of the transponder. Compared to the read-function, this function processes only one page at once. The function is available for all members of the NTAG transponder family.

```
bool NTAG_WritePage(int Page, const byte* Data);
```

Parameters:	
`int Page`	Specify the address of the page to be written. The valid range of this parameter depends on the transponder type.
`const byte* Data`	This parameter holds the data which shall be written to the tag. Note that this function always writes 4 bytes of data, so the minimum array size of `Data` must be at least 4 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.1.3 Fast Read

The Fast Read function reads a number of pages beginning at a starting page from the transponder. The function is supported by NTAG21x and NT3H1xxx transponders.

```
bool NTAG_FastRead(int StartPage, int NumberOfPages, byte* Data);
```

Parameters:	
`int StartPage`	Specify the address of the starting page.
`int NumberOfPages`	Specify the number of pages to be read.
`byte* Data`	This buffer holds the received data from the tag. Take care for proper dimensioning, the buffer size must be at least NumberOfPages * 4.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.2 Miscellaneous functions

19.2.1 Read Counter

This function reads the value of the one-way counter. The function is supported by NTAG21x transponders. Please note that the `NFC_CNT_EN` bit in `ACCESS` configuration byte must be set in order to make this function work.

```
bool NTAG_ReadCounter(int* CounterValue);
```

Parameters:	
`int* CounterValue`	This parameter holds the returned counter value.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.2.2 Read ECC Signature

Use this function to read the factory programmed 32 byte ECC signature, to verify NXP Semiconductors as the silicon vendor. The function is supported by NTAG21x transponders.

```
bool NTAG_ReadSig(byte* ECCSig);
```

Parameters:	
`byte* ECCSig`	This buffer holds the returned ECC signature. The required buffer size is 32 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.2.3 Get Transponder Information

Use this function to retrieve information about the transponder such as product version or storage size. The function is supported by NTAG21x and NT3H1xxx transponders.

```
bool NTAG_GetVersion(byte* Version);
```

Parameters:	
`byte* Version`	This buffer holds 8 bytes of version information.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.2.4 Password Authentication

Use this function for password authentication. For authentication, a 4 bytes password and a 2 bytes acknowledge are required. The function is supported by NTAG21x transponders.

```
bool NTAG_PwdAuth(const byte* Password, const byte* PwdAck);
```

Parameters:	
`const byte* Password`	The 4 bytes password is specified by this parameter.
`const byte* PwdAck`	This buffer holds 2 bytes of Password Acknowledge.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

19.2.5 Select Sector

Use this function to perform a sector select in order o switch between different memory sectors of a NT3H1XXX.

```
bool NTAG_SectorSelect(int Sector);
```

Parameters:	
`int Sector`	Specify the sector to be selected.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20 DESFire Specific Transponder Operations

The memory of a DESFire transponder is organized as a flexible file system. The transponder can hold up to 28 applications and each application may contain up to 32 files of different type and size. Each application can be secured by up to 14 cryptographic keys which are stored in the applications's internal key file. Applications are identified by a number, which must be unambiguous on the transponder. The same rule applies to files within applications, these are identified by numbers which must be unambiguous within the application.
By default, there exists a root-application with the identifier 0x000000 which defines the so-called transponder level. This application cannot hold any files, it is intended to be used for basic administration of the transponder.

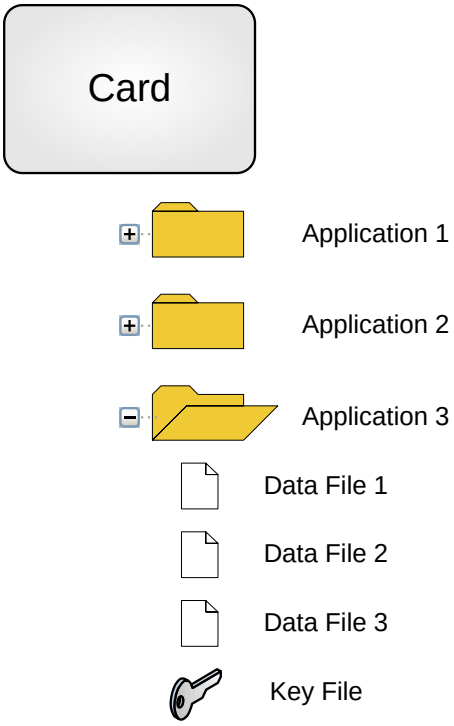


Figure 20.1: DESFire memory layout

A simple use-case could be: Search for a transponder, select the desired application, perform an authentication with the respective key (if required), access data file for read or write operation.

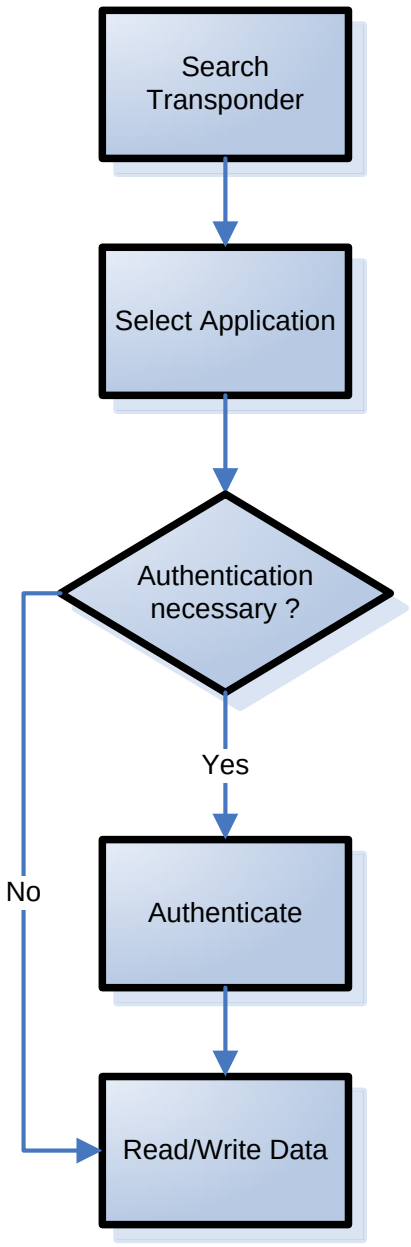


Figure 20.2: Simple way to gain access to the file system

20.1 Security Related Operations

20.1.1 Authenticate

This function shall be used to perform a mutual three pass authentication between reader and transponder. The function supports both 3DES, 3K3DES and AES cryptography. In order to support both the DESFire EV1 transponder family and the older DESFire MF3ICD40, the function incorporates a so-called *Compatible Mode*.
After successful authentication, a session-key is generated which is used for all further cryptographic operations. The authenticated state is invalidated in case of selecting an application, changing the key which was used for the current authentication or a failed authentication.

On transponder level, depending on the security configuration, an authentication with the transponder master key may be required to perform specific operations:

- Gather information on the transponder
- Change the transponder master key
- Change the transponder master key settings
- Create/delete applications

On application level, depending on the configuration, an authentication may be required to perform specific operations:

- Gather information about the application
- Change the keys of the application
- Create/delete files within the application
- Change access rights
- Access data files

```
bool DESFire_Authenticate
    (
    int CryptoEnv,
    int KeyNoTag,
    const byte* Key,
    int KeyByteCount,
    int KeyType,
    int Mode
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. All consecutive operations with the transponder shall be done using the specified environment.
`int KeyNoTag`	Specify the key number that shall be used for authentication. On transponder level, only key 0 is valid for authentication. On application level, one can specify up to 14 keys which can be used for authentication. Both on transponder and application level, key 0 identifies the respective master key.
`const byte* Key`	Specify the key that shall be used for authentication. For 3DES/AES, the key must have a key length of 16 bytes, for 3K3DES the key must have a key length of 24 bytes.
`int KeyByteCount`	Specify the key length of the key. Use one of the predefined constants `DESF_KEYLEN_3DES`, `DESF_KEYLEN_3K3DES` or `DESF_KEYLEN_AES`.
`int KeyType`	Specify the type of the specified key. Use one of the predefined constants `DESF_KEYTYPE_3DES`, `DESF_KEYTYPE_3K3DES` or `DESF_KEYTYPE_AES`. The authentication will be performed according to the specified key type.
`int Mode`	Select either DESFire EV1 ISO-mode authentication or the compatible native DESFire authentication scheme. Use one of the predefined constants `DESF_AUTHMODE_COMPATIBLE` or `DESF_AUTHMODE_EV1`. Note that 3K3DES or AES cryptography cannot be used in compatible mode.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: By default, the initial value of any key is all zeros. E.g. after creation of an application, all keys have this initial value.

Example:

```
// Perform AES-authentication using key 0

const byte Key[16] =
{
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};

if (DESFire_Authenticate(
    CRYPTO_ENV0,
    0,
    Key,
    DESF_KEYLEN_AES,
    DESF_KEYTYPE_AES,
    DESF_AUTHMODE_EV1))
{
    DoSomething();
}
```

20.1.2 Get Key Version

This function can be used to read the current key version of any key that is stored on the transponder. If the selected application is `0x000000`, the command applies to the transponder master key and therefore only key number 0 is valid for querying the key version.

```
bool DESFire_GetKeyVersion
    (
    int CryptoEnv,
    int KeyNo,
    byte* KeyVer
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int KeyNoTag`	Specify the key number that shall be queried.
`byte* KeyVer`	The key version information is returned as one byte by this parameter.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// Query key version of key 0
byte KeyVer;

if (DESFire_GetKeyVersion(CRYPTO_ENV0,0,&KeyVer))
{
    DoSomething();
}
```

20.1.3 Get Key Settings

This function allows to get information on the transponder- or application key settings. Depending on the key settings, a preceding authentication with the respective master key may be required.

```
bool DESFire_GetKeySettings
    (
    int CryptoEnv,
    TDESFireMasterKeySettings* MasterKeySettings
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`TDESFireMasterKeySettings* MasterKeySettings`	This structure receives the queried master key settings.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
`TDESFireKeySettings KeySettings`	8	This member holds the settings of the master key.
`int NumberOfKeys`	32	This member holds the number of available keys. The valid range is 0 to 14.
`int KeyType`	32	This member holds the type of keys. Possible value is one of the predefined constants `DESF_KEYTYPE_3DES`, `DESF_KEYTYPE_3K3DES` or `DESF_KEYTYPE_AES`.

Table 20.1: Definition of `TDESFireMasterKeySettings`

Members	Length (Bits)	Description
`byte AllowChangeMasterKey`	1	If set to 1 the master key is changeable, otherwise it cannot be changed any more.
`byte FreeDirectoryList`	1	If set to 1 no preceding authentication with the master key is required to perform the operations GetFileIDs, GetFileSettings, GetKeySettings (application level) or GetApplicationIDs, GetKeySettings (transponder level). If set to 0, an authentication with the master key is required.
`byte FreeCreateDelete`	1	If set to 1 no preceding authentication with the master key is required to perform the operations CreateFile/DeleteFile (application level) or CreateApplication/DeleteApplication (transponder level). If set to 0, an authentication with the master key is required.
`byte ConfigurationChangeable`	1	If set to 1 the configuration is changeable if authenticated with the master key. If set to 0, the configuration cannot be changed any more.
`byte ChangeKeyAccessRights`	4	This member holds the access rights for changing keys. On transponder level this member is set to 0. 0x0: Authentication with the master key is necessary to change any key. 0x1...0xD: Authentication with the specified key is necessary to change any key. The specified key and the master key can only be changed after authentication with the master key. 0xE: Authentication with the key to be changed is necessary to change the key. 0xF: All keys except the master key are frozen.

Table 20.2: Definition of `TDESFireKeySettings`

Example:

```
// Query key settings of application 0x123456

TDESFireMasterKeySettings MasterKeySettings;

if (DESFire_SelectApplication(0x123456))
{
    if (DESFire_GetKeySettings(CRYPTO_ENV0,&MasterKeySettings))
    {
        DoSomething(MasterKeySettings);
    }
}
```

20.1.4 Change Key Settings

This function allows to change the transponder- or application master key settings. The respective master key settings can only be changed, if the bit `ConfigurationChangeable` of the current key settings was not

cleared before. In order to change the key settings, a preceding authentication with the respective master key is required in general.

```
bool DESFire_ChangeKeySettings
    (
    int CryptoEnv,
    const TDESFireMasterKeySettings* MasterKeySettings
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`const TDESFireMasterKeySettings* MasterKeySettings`	This structure holds the new master key settings. See chapter *Get Key Settings* for details.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.1.5 Change Key

This function allows to change a key. The respective key settings define (see chapter *Get Key Settings*) whether changing of a key is permitted or not and which key must be used for authentication before calling this function.

```
bool DESFire_ChangeKey
    (
    int CryptoEnv,
    int KeyNo,
    const byte* OldKey,
    int OldKeyByteCount,
    const byte* NewKey,
    int NewKeyByteCount,
    byte KeyVersion,
    const TDESFireMasterKeySettings* MasterKeySettings
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int KeyNo`	Specify the key number that shall be changed.
`const byte* OldKey`	Specify the old key.
`int OldKeyByteCount`	Specify the length of the old key in bytes.
`const byte* NewKey`	Specify the new key.
`int NewKeyByteCount`	Specify the length of the new key in bytes.
`byte KeyVersion`	Specify the key version of the new key.
`const TDESFireMasterKeySettings* MasterKeySettings`	This structure holds the current master key settings. See chapter *Get Key Settings* for details.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// Change key 1 of application 0x123456

const byte oldKey[16] =
{
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
const byte newKey[16] =
{
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};
TDESFireMasterKeySettings MasterKeySettings;

if (!DESFire_SelectApplication(0x123456))
{
    return; // Error selecting application
}
if (!DESFire_GetKeySettings(CRYPTO_ENV0, &MasterKeySettings))
{
    return; // Error gathering key settings
}
if (MasterKeySettings.KeySettings.ChangeKeyAccessRights == 0)
{
    // Authenticate with master key
    if (!DESFire_Authenticate(
        CRYPTO_ENV0,
        0,
        oldKey,
        DESF_KEYLEN_AES,
        DESF_AUTHMODE_EV1))
    {
        return; // Authentication error
    }
    if (!DESFire_ChangeKey(
```

```
        CRYPTO_ENV0,
        1,
        oldKey,
        DESF_KEYLEN_AES,
        newKey,
        DESF_KEYLEN_AES,
        0x20,
        &MasterKeySettings))
    {
        return; // Error changing key 1
    }
}
```

20.2 Transponder Related Operations

20.2.1 Create Application

This function allows to create a new application on the transponder. Depending on the security settings of the transponder, a preceding authentication with the transponder master key may be required, see chapter *Get Key Settings* for details.

```
bool DESFire_CreateApplication
    (
    int CryptoEnv,
    int AID,
    const TDESFireMasterKeySettings* MasterKeySettings
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int AID`	Specify the Application ID of the new application to be created. The AID consists of 24 bit, its value must be unique on the transponder. The value 0x000000 is reserved for the root application.
`const TDESFireMasterKeySettings* MasterKeySettings`	This structure holds the master key settings of the new application. See chapter *Get Key Settings* for details.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// Create application 0x123456

TDESFireMasterKeySettings MasterKeySettings;

MasterKeySettings.KeySettings.AllowChangeMasterKey    = true;
MasterKeySettings.KeySettings.FreeDirectoryList       = true;
MasterKeySettings.KeySettings.FreeCreateDelete        = true;
MasterKeySettings.KeySettings.ConfigurationChangeable = true;
```

```
MasterKeySettings.KeySettings.ChangeKeyAccessRights   = 0x0;
MasterKeySettings.NumberOfKeys = 2;
MasterKeySettings.KeyType      = DESF_KEYTYPE_AES;

if (DESFire_CreateApplication(
    CRYPTO_ENV0,
    0x123456,
    &MasterKeySettings))
{
        DoSomething();
}
```

20.2.2 Delete Application

This function allows to delete an existing application on the transponder. Depending on the security settings of the transponder, a preceding authentication with the transponder master key may be required, see chapter *Get Key Settings* for details.

```
bool DESFire_DeleteApplication
    (
    int CryptoEnv,
    int AID
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int AID`	Specify the Application ID of the application that shall be deleted. The AID consists of 24 bit. The value 0x000000 is reserved for the root application hence this AID cannot be deleted.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.2.3 Get Application IDs

This function allows to list all application IDs that exist on the transponder. Depending on the security settings of the transponder, a preceding authentication with the transponder master key may be required, see chapter *Get Key Settings* for details.

```
bool DESFire_GetApplicationIDs
    (
    int CryptoEnv,
    int* AIDs,
    int* NumberOfAIDs,
    int MaxAIDCnt
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int* AIDs`	After successful completion of this function, this parameter holds a list of the retrieved application IDs.
`int* NumberOfAIDs`	This parameter holds the number of retrieved application IDs.
`int MaxAIDCnt`	Specify the maximum number of application IDs, that can be stored in the array `AIDs`. Note: Up to 28 applications can be stored on a DESFire transponder, so take care for proper dimensioning of the array `AIDs`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// List applications stored on the transponder

int AIDList[28];
int NumberOfAIDs;

if (DESFire_GetApplicationIDs(
    CRYPTO_ENV0,
    AIDList,
    &NumberOfAIDs,
    sizeof(AIDList)/sizeof(int)))
{
    DoSomething(AIDList,NumberOfAIDs);
}
```

20.2.4 Select Application

This function is used to select an application in order to perform further operations such as reading or writing. Depending on the security settings of the selected application, an authentication with one of the application's keys may be required after selection.

```
bool DESFire_SelectApplication
    (
    int CryptoEnv,
    int AID
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int AID`	This parameter holds the application ID of the application to be selected.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.2.5 Format Transponder

Calling this function results in formatting the transponder. This means, all applications including their files and keys are destroyed and the occupied memory space is released for future use. For proper usage, a preceding authentication with the transponder master key is required.

```
bool DESFire_FormatTag
    (
    int CryptoEnv
    );
```

20.2.6 Get Transponder Information

This function can be used to gather detailed information about the DESFire transponder regarding hardware and software version.

```
bool DESFire_GetVersion
    (
    int CryptoEnv,
    TDESFireVersion* Version
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`TDESFireVersion* Version`	This structure receives the queried manufacturing related information.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
`TDESFireTagInfo HWInfo`	80	This member holds the hardware related version information.
`TDESFireTagInfo SWInfo`	80	This member holds the software related version information.
`TDESFireProdInfo ProdInfo`	112	This member holds manufacturing specific information.

Table 20.3: Definition of `TDESFireVersion`

Members	Length (Bits)	Description
byte VendorID	8	Codes the vendor ID (0x04 stands for NXP).
byte Type	8	Codes the type (here 0x01).
byte SubType	8	Codes the subtype(here 0x01).
byte VersionMajor	8	Codes the major version number.
byte VersionMinor	8	Codes the minor version number.
uint32_t StorageSize	32	Size of EEPROM in bytes.
byte CommunicationProtocol	8	Codes the communication protocol type (here 0x05 means ISO14443-3 and -4).

Table 20.4: Definition of TDESFireTagInfo

Members	Length (Bits)	Description
byte UID[7]	56	This member holds the unique serial number. If the transponder is configured to Random ID, the UID is set to 0x00.
byte ProdBatchNumber[5]	40	Codes the production batch number.
byte CalendarWeekOfProduction	8	Codes the calendar week of production.
byte YearOfProduction	8	Codes the year of production.

Table 20.5: Definition of TDESFireProdInfo

20.2.7 Get Available Memory Space

This function allows to gather the available memory space of the transponder. A preceding authentication is not required.

```
bool DESFire_FreeMem
    (
    int CryptoEnv,
    int* FreeMemory
    );
```

Parameters:

int CryptoEnv	Specify a cryptographic environment by this parameter. The valid range is CRYPTO_ENV0 to CRYPTO_ENV3, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
int* FreeMemory	After successful completion of this function, the available memory size in bytes is returned by this parameter.
Return:	If the operation was successful, the return value is true, otherwise it is false.

20.2.8 Get Card UID

This function allows to retrieve the card UID in case of random ID. A preceding authentication with any key is required prior calling this function.

```
bool DESFire_GetUID
    (
    int CryptoEnv,
    byte* UID,
    int* Length,
    int BufferSize
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`byte* UID`	After successful completion of this function, the real card UID is returned by this parameter. Note: The UID usually occupies 7 bytes, so take care for proper dimensioning of the array `UID`.
`int* Length`	The length in bytes of the UID is returned by this parameter.
`int BufferSize`	This parameter specifies the size of the array `UID` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.2.9 Set Transponder Configuration

20.2.9.1 Disable Format Tag

When this function is called, formatting the transponder is not possible any more (see chapter *Format Transponder*). A preceding authentication with the transponder master key is required prior calling this function. Note: Disabling tag formatting cannot be reset any more.

```
bool DESFire_DisableFormatTag
    (
    int CryptoEnv
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.2.9.2 Enable Random ID

When this function is called, the transponder is turned into Random ID mode, this means the real UID can only be retrieved by authenticating to the transponder and calling the function *Get Card UID*. A preceding authentication with the transponder master key is required prior calling this function. Note: Setting the transponder to Random ID mode cannot be reset any more.

```
bool DESFire_EnableRandomID
    (
    int CryptoEnv
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.2.9.3 Set Default Key

This function can be used to specify the default key, which is applied when e.g. a new application is created on the transponder. By default, keys are initialized to 0x00. A preceding authentication with the transponder master key is required prior calling this function.

```
bool DESFire_SetDefaultKey
    (
    int CryptoEnv,
    const byte* Key,
    int KeyByteCount,
    byte KeyVersion
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`const byte* Key`	This parameter specifies the new default key.
`int KeyByteCount`	This parameter specifies the length of the new default key in bytes. Use one of the predefined constants `DESF_KEYLEN_3DES`, `DESF_KEYLEN_3K3DES` or `DESF_KEYLEN_AES`.
`byte KeyVersion`	This parameter specifies the default key version.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.2.9.4 Set User-defined Answer To Select (ATS)

This function can be used to specify a user-defined Answer To Select (ATS) which is returned by the transponder after RATS. Changing the ATS to a non-default value shall only be carried out by experts as a ATS longer than 16 bytes could cause problems with readers that support only frame sizes of max. 16 bytes. The ATS must be formatted as follows: TL T0 TA TB TC + Historical bytes. The default ATS of DESFire EV1 is TL=0x06, T0=0x75, TA=0x77, TB=0x81, TC=0x02, Historical Bytes=0x80.

```
bool DESFire_SetATS
    (
    int CryptoEnv,
    const byte* ATS,
    int Length
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`const byte* ATS`	This parameter specifies the new ATS.
`int Length`	This parameter specifies the length of the new ATS in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.3 Application Related Operations

This section deals with file handling within an application of a DESFire transponder. An application can hold three different basic file types: Data files, Value files and Record Files. Data files are available either with or without integrated backup-mechanism, Value files and Record files always incorporate integrated backup. There exist two types of record files: Linear record files and Cyclic Record Files.
Some functions for file handling are using the data structure `TDESFireFileSettings` which defines all relevant file settings. See the following tables for reference:
Coding of access rights:
Every file holds four different access rights, each access right is coded in one nibble. These four nibbles are concatenated and form the 16 bit variable `AccessRights`.

One nibble codes 16 possible values. If it codes a number between 0 and 13, this references a certain key number within the application.
If the number is 14, this means "free" access so there is no authentication necessary to perform the respective operation on the file. In case of coding the number 15, this means "deny" access.

20.3.1 Create File

This section deals with the creation of new files within applications. Depending on the specified file type, the file is either created with or without integrated backup-mechanism. Each file requires an unambiguous identifier which is coded in one byte in the range from 0x00 to 0x1F. During creation of the file, the level of security is defined in the communication settings. Communication can be either plain, secured by MAC or

Members	Length (Bits)	Description
byte FileType	8	This member defines the file type. Possible values are: DESF_FILETYPE_STDDATAFILE, DESF_FILETYPE_BACKUPDATAFILE, DESF_FILETYPE_VALUEFILE, DESF_FILETYPE_LINEARRECORDFILE, DESF_FILETYPE_CYCLICRECORDFILE.
byte CommSet	8	This member defines the communication settings between reader and transponder when the file is accessed. Possible values are: DESF_COMMSET_PLAIN, DESF_COMMSET_PLAIN_MACED, DESF_COMMSET_FULLY_ENC
uint16_t AccessRights	16	This member holds the access rights.
union TDESFireSpecificFileInfo SpecificFileInfo	32 to 128	This member holds file type specific information.

Table 20.6: Definition of TDESFireFileSettings

15...12	11...8	7...4	3...0
Read Access	Write Access	Read/Write Access	Change Access Rights

Table 20.7: Coding of AccessRights

Members	Length (Bits)	Description
struct TDESFireDataFileSettings DataFileSettings	32	Definition of data file settings.
struct TDESFireValueFileSettings ValueFileSettings	128	Definition of value file settings.
struct TDESFireRecordFileSettings RecordFileSettings	96	Definition of record file settings.

Table 20.8: Definition of union TDESFireSpecificFileInfo

Members	Length (Bits)	Description
uint32_t FileSize	32	Definition of the data file size.

Table 20.9: Definition of struct TDESFireDataFileSettings

fully enciphered. Furthermore, the access rights are assigned to certain keys held by the application. Depending on the security settings of the application, a preceding authentication with the application master key may be required, see chapter *Get Key Settings* for details.

Members	Length (Bits)	Description
`uint32_t LowerLimit`	32	Definition of the lower limit which must not be passed by a debit operation.
`uint32_t UpperLimit`	32	Definition of the upper limit which must not be passed by a credit operation.
`uint32_t LimitedCreditValue`	32	Definition of the initial value of the file at file creation.
`TValueFileOptions ValueFileOptions`	32	Specific options for value files.

Table 20.10: Definition of `struct TDESFireValueFileSettings`

Members	Length (Bits)	Description
`byte LimitedCreditEnable`	1	Limited Credit feature enabled or disabled.
`byte FreeGetValue`	1	Free read access enabled or disabled.

Table 20.11: Definition of `struct TValueFileOptions`

Members	Length (Bits)	Description
`uint32_t RecordSize`	32	Definition of the size of one single record in bytes.
`uint32_t MaxNumberOfRecords`	32	Definition of the maximum number of records.
`uint32_t CurrentNumberOfRecords`	32	Definition of the current number of records. This member is ignored at file creation.

Table 20.12: Definition of `struct TDESFireRecordFileSettings`

```
bool DESFire_CreateDataFile
    (
    int CryptoEnv,
    int FileNo,
    const TDESFireFileSettings* FileSettings
    );

bool DESFire_CreateValueFile
    (
    int CryptoEnv,
    int FileNo,
    const TDESFireFileSettings* FileSettings
    );

bool DESFire_CreateRecordFile
    (
    int CryptoEnv,
    int FileNo,
    const TDESFireFileSettings* FileSettings
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the file ID. If the ID already exists within the application, this results in an error.
`const TDESFireFileSettings* FileSettings`	This member holds the file settings. See description of `TDESFireFileSettings` for details.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// Create new standard data file (without backup)
// in application 0x123456

TDESFireFileSettings FileSettings;
int FileID;

if (DESFire_SelectApplication(0x123456))
{
    // We create a standard data file
    FileSettings.FileType = DESF_FILETYPE_STDDATAFILE;
    // Communication between reader and tag is fully enciphered
    FileSettings.CommSet  = DESF_COMMSET_FULLY_ENC;
    // Read Access     : Key 1
    // Write Access    : Key 2
    // Read/Write      : Key 3
    // Change Settings : Key 4
    FileSettings.AccessRights = 0x1234;
    // File size shall be 512 bytes
    FileSettings.SpecificFileInfo.DataFileSettings.FileSize = 512;
    // Assign an identifier to the file
    FileID = 0x12;
    if (DESFire_CreateDataFile(CRYPTO_ENV0, FileID, &FileSettings))
    {
        DoSomething();
    }
}
```

20.3.2 Delete File

This function allows to permanently deactivate a file within an application. This means, the allocated memory is not released for further usage, only the file number can be re-used for creating a new file. In order to re-use the memory of deleted files, this requires formatting the transponder but this leads to permanent loss of any application data. Depending on the security settings of the application, a preceding authentication with the application master key may be required, see chapter *Get Key Settings* for details.

```
bool DESFire_DeleteFile
    (
    int CryptoEnv,
    int FileNo
```

```
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the file which shall be deleted. If the ID doesn't exist within the application, this results in an error.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.3.3 Get File IDs

This function allows to list all file IDs that exist within the currently selected application. Each file ID is coded in one byte in the range from 0x00 to 0x1F. Duplicate values are not possible as each file must have an unambiguous identifier. Depending on the security settings of the application, a preceding authentication with the application master key may be required, see chapter *Get Key Settings* for details.

```
bool DESFire_GetFileIDs
    (
    int CryptoEnv,
    byte* FileIDList,
    int* FileIDCount,
    int MaxFileIDCount
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`byte* FileIDList`	After successful completion of this function, this parameter holds a list of the retrieved file IDs.
`int* FileIDCount`	This parameter holds the number of retrieved file IDs.
`int MaxFileIDCount`	Specify the maximum number of file IDs, that can be stored in the array `FileIDList`. Note: Up to 32 files can be stored within an application, so take care for proper dimensioning of the array `FileIDList`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:
See chapter *Get File Settings* for a comprehensive example.

20.3.4 Get File Settings

This function allows to query the file settings of an existing file within an application. The returned information depends on the type of the file. Depending on the security settings of the application, a preceding authentication with the application master key may be required, see chapter *Get Key Settings* for details.

```
bool DESFire_GetFileSettings
    (
    int CryptoEnv,
    int FileNo,
    TDESFireFileSettings* FileSettings
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the file ID which shall be queried.
`TDESFireFileSettings* FileSettings`	This member holds the returned file settings. See description of `TDESFireFileSettings` for details.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// Query file settings of all files in application 0x123456

TDESFireFileSettings FileSettings;

// An application can hold up to 32 files
byte FileIDList[32];
int FileIDCount;

int i;

if (DESFire_SelectApplication(0x123456))
{
    // Gather a list of present file IDs
    if (DESFire_GetFileIDs(
        CRYPTO_ENV0,
        FileIDList,
        &FileIDCount,
        sizeof(FileIDList)))
    {
        for (i=0; i<FileIDCount; i++)
        {
            // Query the settings of each file
            if (DESFire_GetFileSettings(
                CRYPTO_ENV0,
                FileIDList[i],
                &FileSettings))
            {
                switch(FileSettings.FileType)
                {
                case DESF_FILETYPE_STDDATAFILE:
                    DoSomething();
                    break;
                case DESF_FILETYPE_VALUEFILE:
                    DoSomethingElse();
                    break;
                }
            }
```

```
        }
    }
}
```

20.3.5 Change File Settings

This function allows to change the access parameters such as communication settings and access rights of an existing file. Depending on the actual change access rights of the file, authentication with the respective key has to be performed before calling this function. Furthermore, the change access right must be different from "deny". See *Coding of Access Rights* for details.

```
bool DESFire_ChangeFileSettings
    (
    int CryptoEnv,
    int FileNo,
    int NewCommSet,
    int OldAccessRights,
    int NewAccessRights
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the file ID whose settings shall be changed.
`int NewCommSet`	Specify the new communication settings. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
`int OldAccessRights`	Specify the current Access Rights of the file.
`int NewAccessRights`	Specify the new Access Rights of the file.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.4 File Related Operations

20.4.1 Data Files

20.4.1.1 Read Data

This function shall be used to access a standard or backup data file in order to read from it. Depending on the file's access rights, a preceding authentication with the read or read/write key has to be done, see *Coding of Access Rights* for details. The function allows segmented access, this means the user is able to either read the entire file or only a part starting at a user-defined offset.

```
bool DESFire_ReadData
    (
    int CryptoEnv,
```

```
    int FileNo,
    byte* Data,
    int Offset,
    int Length,
    int CommSet
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the file that shall be read.
`byte* Data`	After successful completion of this function, the buffer referred by this parameter holds the data which was read from the transponder. Take care for adequate dimensioning.
`int Offset`	Specify the starting address for reading. The valid range of this parameter is 0x000000 to FileSize - 1. In case of address-range violation, the function returns with an error.
`int Length`	Specify the length of data that shall be read. The valid range of this parameter is FileSize - Offset. In case of address-range violation, the function returns with an error.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Example:

```
// Read data file 0x12 which is located in application 0x123456

TDESFireFileSettings FileSettings;

int ReadAccess;

// This is the buffer that receives the data to be read
byte Data[512];

// If an authentication is necessary, we assume this would be
// the key that gives read access
const byte KeyRead[16] =
{
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};

if (!DESFire_SelectApplication(CRYPTO_ENV0, 0x123456))
    return;    // Error

// Gather file settings
if (!DESFire_GetFileSettings(CRYPTO_ENV0, 0x12, &FileSettings))
    return;    // Error
```

```
// Read access rights are located in the highest nibble of
// FileSettings.AccessRights
ReadAccess = (FileSettings.AccessRights >> 12) & 0x000F;

switch (ReadAccess)
{
case 15:    // Access denied
    return;
case 14:    // Free access
    break;
default:
    // Authenticate with the "reading-key"
    if (!DESFire_Authenticate(
        CRYPTO_ENV0,
        ReadAccess,
        KeyRead,
        DESF_KEYLEN_AES,
        DESF_KEYTYPE_AES,
        DESF_AUTHMODE_EV1))
        return;    // Error
}

// Check size of reading buffer
if (FileSettings.SpecificFileInfo.DataFileSettings.FileSize >
    sizeof(Data))
    return;   // Buffer size not enough

// Read entire data file
if (DESFire_ReadData(
    CRYPTO_ENV0,
    0x12,
    Data,
    0,
    FileSettings.SpecificFileInfo.DataFileSettings.FileSize,
    FileSettings.CommSet))
{
    DoSomething();
}
```

20.4.1.2 Write Data

This function shall be used to access a standard or backup data file in order to write to it. Depending on the file's access rights, a preceding authentication with the write or read/write key has to be done, see *Coding of Access Rights* for details. The function allows segmented access, this means the user is able to either rewrite the entire file or only a part starting at a user-defined offset.

```
bool DESFire_WriteData
    (
    int CryptoEnv,
    int FileNo,
    const byte* Data,
    int Offset,
    int Length,
    int CommSet
```

```
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the file that shall be written.
`const byte* Data`	The buffer referred by this parameter holds the data which is written to the file.
`int Offset`	Specify the starting address for writing. The valid range of this parameter is 0x000000 to FileSize - 1. In case of address-range violation, the function returns with an error.
`int Length`	Specify the length of data that shall be written. The valid range of this parameter is FileSize - Offset. In case of address-range violation, the function returns with an error.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: If data is written to a Backup Data File, it is necessary to validate the written data with the function *Commit Transaction*. Calling the function *Abort Transaction* will invalidate all changes.

Example:

```
// Write to data file 0x12 which is located in application 0x123456

TDESFireFileSettings FileSettings;

int WriteAccess;

// This is the buffer that holds the data to be written
const byte Data[] =
{
    0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08
};

// If an authentication is necessary, we assume this would be
// the key that gives write access
const byte KeyWrite[16] =
{
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};

if (!DESFire_SelectApplication(CRYPTO_ENV0, 0x123456))
    return;    // Error

// Gather file settings
if (!DESFire_GetFileSettings(CRYPTO_ENV0, 0x12, &FileSettings))
    return;    // Error
```

```
// Write access rights are located in bits 11...8 of
// FileSettings.AccessRights
WriteAccess = (FileSettings.AccessRights >> 8) & 0x000F;

switch (WriteAccess)
{
case 15:    // Access denied
    return;
case 14:    // Free access
    break;
default:
    // Authenticate with the "writing-key"
    if (!DESFire_Authenticate(
        CRYPTO_ENV0,
        WriteAccess,
        KeyWrite,
        DESF_KEYLEN_AES,
        DESF_KEYTYPE_AES,
        DESF_AUTHMODE_EV1))
        return;    // Error
}

// Check size of file
if (FileSettings.SpecificFileInfo.DataFileSettings.FileSize <
    sizeof(Data))
    return;   // File size not enough

// Write to data file
if (DESFire_WriteData(
    CRYPTO_ENV0,
    0x12,
    Data,
    0,
    sizeof(Data),
    FileSettings.CommSet))
{
    DoSomething();
}
```

20.4.2 Value Files

20.4.2.1 Get Value

This function allows to read the current value from a Value File. Depending on the file's access rights, a preceding authentication with the read, write or read/write key has to be done, see *Coding of Access Rights* for details.

```
bool DESFire_GetValue
    (
    int CryptoEnv,
    int FileNo,
    int* Value,
    int CommSet
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the Value File whose value shall be queried.
`int* Value`	After successful completion of this function, this parameter holds the value which was read from the file.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.4.2.2 Debit

This function allows to decrease a value stored in a Value File. The function requires a preceding authentication with the read, write or read/write key, see *Coding of Access Rights* for details. The value modifications of *Credit*, *Debit* and *Limited Credit* functions are cumulated until the function *Commit Transaction* is called.
If the *Limited Credit feature* is enabled, the new limit for a subsequent *Limited Credit* function call is set to the sum of *Debit* modifications within one transaction before calling *Commit Transaction*. This assures, that a *Limited Credit* can not re-book more values than a debiting transaction deducted before.

```
bool DESFire_Debit
    (
    int CryptoEnv,
    int FileNo,
    const int Value,
    int CommSet
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the Value File that shall be debited.
`const int Value`	The value stored in the value file will be decreased by this parameter.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: After modifying value files, it is necessary to validate the transaction with the function *Commit Transaction*. Calling the function *Abort Transaction* will invalidate all changes.

20.4.2.3 Credit

This function allows to increase a value stored in a Value File. The function requires a preceding authentication with the read/write key, see *Coding of Access Rights* for details. The value modifications of *Credit*, *Debit* and *Limited Credit* functions are cumulated until the function *Commit Transaction* is called.
If the *Limited Credit feature* is enabled, this function cannot be used. Use the function *Limited Credit* instead.

```
bool DESFire_Credit
    (
    int CryptoEnv,
    int FileNo,
    const int Value,
    int CommSet
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the Value File that shall be credited.
`const int Value`	The value stored in the value file will be increased by this parameter.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: After modifying value files, it is necessary to validate the transaction with the function *Commit Transaction*. Calling the function *Abort Transaction* will invalidate all changes.

20.4.2.4 Limited Credit

This function allows a limited increase of a value stored in a Value File without having full read/write permissions to the file. This feature can only be used if it has been enabled during file creation. The function requires a preceding authentication with the write or read/write key, see *Coding of Access Rights* for details. The value modifications of *Credit*, *Debit* and *Limited Credit* functions are cumulated until the function *Commit Transaction* is called.
After calling this function, the new limit is set to 0, regardless of the amount which has been re-booked. Hence, this function can only be used once after a Debit transaction.

```
bool DESFire_LimitedCredit
    (
    int CryptoEnv,
    int FileNo,
    const int Value,
    int CommSet
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the Value File that shall be credited.
`const int Value`	The value stored in the value file will be increased by this parameter. It is limited to the sum of Debit operations on this value file within the most recent transaction containing at least one Debit.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: After modifying value files, it is necessary to validate the transaction with the function *Commit Transaction*. Calling the function *Abort Transaction* will invalidate all changes.

20.4.3 Record Files

20.4.3.1 Read Records

Use this function to read out a set of complete records from a Record File. The function requires a preceding authentication with the read or read/write key, see *Coding of Access Rights* for details.

```
bool DESFire_ReadRecords
    (
    int CryptoEnv,
    int FileNo,
    byte* RecordData,
    int* RecDataByteCnt,
    int Offset,
    int NumberOfRecords,
    int RecordSize,
    int CommSet
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the file that shall be read.
`byte* RecordData`	After successful completion of this function, the buffer referred by this parameter holds the data which was read from the transponder. Take care for adequate dimensioning.
`int* RecDataByteCnt`	The total number of bytes read from the transponder is represented by this parameter.
`int Offset`	Specify the offset of the newest record to be read out. The valid range of this parameter is 0x000000 to number of existing records - 1. In case of 0x000000 the latest record is read out.
`int NumberOfRecords`	Specify the number of records to be read out.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.4.3.2 Write Record

Use this function to write data to a Record File. The function requires a preceding authentication with the write or read/write key, see *Coding of Access Rights* for details. In order to validate writing, a call of *Commit Transaction* becomes necessary. If writing is not validated, a new WriteRecord command writes to the already created record.

```
bool DESFire_WriteRecord
    (
    int CryptoEnv,
    int FileNo,
    const byte* Data,
    int Offset,
    int Length,
    int CommSet
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the file that shall be read.
`const byte* Data`	This buffer holds the record data to be written.
`int Offset`	Specify the offset in bytes within one single record. The valid range of this parameter is 0x000000 to record size - 1.
`int Length`	Specify the length of data to be written. The parameter has to be in the range from 0x000001 to record size - offset.
`int CommSet`	Specify the communication settings. The communication settings must match to the actual settings of the file. Possible values are: `DESF_COMMSET_PLAIN`, `DESF_COMMSET_PLAIN_MACED`, `DESF_COMMSET_FULLY_ENC`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.4.3.3 Clear Record File

Use this function to reset a Record File to the empty state. The function requires a preceding authentication with the read/write key, see *Coding of Access Rights* for details. After execution of the function, a call of *Commit Transaction* becomes necessary.

```
bool DESFire_ClearRecordFile(int CryptoEnv, int FileNo);
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
`int FileNo`	Specify the ID of the file that shall be cleared.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.4.4 Commit Transaction

This function allows to validate all previous modifications on files with integrated backup mechanism such as Backup Data Files, Value Files and Record Files. When a transaction has been finished, this is usually the last called function; if this step was omitted, any changes would be lost if a different application is selected or the transponder is removed from the RF-field.

```
bool DESFire_CommitTransaction
    (
    int CryptoEnv
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

20.4.5 Abort Transaction

This function allows to discard all previous modifications on files with integrated backup mechanism such as Backup Data Files, Value Files and Record Files.

```
bool DESFire_AbortTransaction
    (
    int CryptoEnv
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants. Usually the same environment is specified that was used for authentication.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

21 SAM AV1/AV2

Before using one of the following functions, a NXP SAM AV1/AV2 card must have been inserted into one of the available SAM slots. When powering up, TWN4 scans the slots for SAM cards, so a correctly inserted SAM card is detected automatically for later use.

21.1 Host Authentication

This function shall be used to perform a mutual three pass authentication between host (reader) and the SAM AV1/AV2 card. The function supports both 3DES and AES cryptography. Depending on security settings of the SAM card, the authentication might be necessary in order to perform different security related actions afterwards.

```
bool SAMAVx_AuthenticateHost
    (
    int CryptoEnv,
    int KeyNo,
    const byte* Key,
    int KeyByteCount,
    int KeyType
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int KeyNo`	Specify the key number that shall be used for authentication.
`const byte* Key`	Specify the key that shall be used for authentication. For 3DES/AES, the key must have a key length of 16 bytes.
`int KeyByteCount`	Specify the key length of the key. Use one of the predefined constants `DESF_KEYLEN_3DES` or `DESF_KEYLEN_AES`.
`int KeyType`	Specify the type of the specified key. Use one of the predefined constants `DESF_KEYTYPE_3DES` or `DESF_KEYTYPE_AES`. The authentication will be performed according to the specified key type.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

21.2 Query Key Entry

Use this function to query information about a key entry on the SAM card.

```
bool SAMAVx_GetKeyEntry(int KeyNo, TSAMAVxKeyEntryData* KeyEntryData);
```

Parameters:

`int KeyNo`	Specify the key number that shall be used for authentication.
`TSAMAVxKeyEntryData* KeyEntryData`	The key entry is returned by this parameter.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
`byte VersionKeyA`	8	This member holds the version of Key A.
`byte VersionKeyB`	8	This member holds the version of Key B.
`byte VersionKeyC`	8	This member holds the version of Key C.
`uint32_t DF_AID`	32	This member holds the associated DESFire AID.
`byte DF_KeyNo`	8	This member holds the associated DESFire key number.
`byte KeyNoCEK`	8	This member holds the key number of the change entry key.
`byte KeyNoVCEK`	8	This member holds the key version of the change entry key.
`byte RefNoKUC`	8	This member holds the number of the associated Key Usage Counter.
`uint16_t SET`	16	This member holds the configuration settings of the key entry.

Table 21.1: Definition of `TSAMAVxKeyEntryData`

22 ISO15693 Specific Transponder Operations

22.1 Generic ISO15693 Command

This function can be used for ISO15693 specific transponder operations which are not covered by high-level system functions.

```
bool ISO15693_GenericCommand
    (
    byte Flags,
    byte Command,
    byte* Data,
    int* Length,
    int BufferSize
    );
```

Parameters:

`byte Flags`	Specify the ISO15693 flags. Note: The flags regarding RF-communication are set automatically, so by default one may assign 0x00 to this parameter.
`byte Command`	Command code.
`byte* Data`	This parameter works as Input/Output-buffer. All additional parameters which are sent to the transponder are passed within this buffer. This buffer is also used for data returned from the transponder.
`int* Length`	This parameter works as Input/Output-variable. It holds the payload-length of `Data` in the directions *Reader→Tag* and *Tag→Reader*.
`int BufferSize`	This parameter holds the array-size of `Data` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

22.2 Gather Tag Specific Information

22.2.1 Get System Information

This function returns more in-depth information of the tag. The function is available in two versions (Protocol Extension flag set or reset), as some tag types like ST 24LR16/64 require the Protocol Extension flag to be set for proper operation.

```
bool ISO15693_GetSystemInformation
    (
    TISO15693_SystemInfo* SystemInfo
    );
```

Members	Length (Bits)	Description
byte DSFID_Present	1	Set to 1 if DSFID is present
byte AFI_Present	1	Set to 1 if AFI is present
byte VICC_Memory_Size_Present	1	Set to 1 if BlockSize and Number_of_Blocks are present
byte IC_Reference_Present	1	Set to 1 if IC_Reference is present
byte Res1	4	Reserved for future use
byte UID[8]	64	Unique Identifier
byte DSFID	8	Data Storage Format Identifier
byte AFI	8	Application Family Identifier
byte BlockSize	8	Size of one data block in bytes
uint16_t Number_of_Blocks	16	Number of available blocks
byte IC_Reference	8	Meaning defined by the IC manufacturer

Table 22.1: Definition of TISO15693_SystemInfo

```
bool ISO15693_GetSystemInformationExt
    (
    TISO15693_SystemInfo* SystemInfo
    );
```

Parameters:

TISO15693_SystemInfo* SystemInfo	Pointer to the structure which receives the System Information.
Return:	If the operation was successful, the return value is true, otherwise it is false.

Remark: As the GetSystemInformation command is no mandatory ISO15693 command, it is not implemented in all tag types available on the market.

Definition	Value	Manufacturer	Tag Type
ISO15693_TAGTYPE_ICODESL2	0x00	NXP	ICode SL2
ISO15693_TAGTYPE_ICODESL2S	0x01		ICode SL2S
ISO15693_TAGTYPE_UNKNOWNNXP	0x0F		Unknown
ISO15693_TAGTYPE_TAGITHFIPLUSINLAY	0x10	TI	Tag-It HFI Plus Inlay
ISO15693_TAGTYPE_TAGITHFIPLUSCHIP	0x11		Tag-It HFI Plus Chip
ISO15693_TAGTYPE_TAGITHFISTD	0x12		Tag-It HFI Standard
ISO15693_TAGTYPE_TAGITHFIPRO	0x13		Tag-It HFI Pro
ISO15693_TAGTYPE_UNKNOWNTI	0x1F		Unknown
ISO15693_TAGTYPE_UNKNOWNST	0x4F	ST	Unknown
ISO15693_TAGTYPE_SRF55V02P	0x50	Infineon	SRF55V02P
ISO15693_TAGTYPE_SRF55V10P	0x51		SRF55V10P
ISO15693_TAGTYPE_SRF55V02S	0x52		SRF55V02S
ISO15693_TAGTYPE_SRF55V10S	0x53		SRF55V10S
ISO15693_TAGTYPE_UNKNOWNINFINEON	0x5F		Unknown
ISO15693_TAGTYPE_UNKNOWN	0xFF	Unknown	Unknown ISO15693

Table 22.2: Retrievable tag types from UID

22.2.2 Get Tag Type

The ISO15693 API incorporates two methods to determine the tag type, either by analysing the UID or the System Information structure.

22.2.2.1 Get Tag Type From UID

This function can be used to determine the tag type of ISO15693 compliant transponders if only the UID is available.

```
int ISO15693_GetTagTypeFromUID
    (
    byte* UID
    );
```

Parameters:

`byte* UID`	This parameter holds the UID. Watch for the correct byte order; UID[0] shall have the value 0xE0
Return:	The return-value is the determined tag-type which is represented by one of the constants in the table below.

22.2.2.2 Get Tag Type From System Information

This function can be used to determine the tag type of ISO15693 compliant transponders if the System Information is available.

```
int ISO15693_GetTagTypeFromSystemInfo
    (
    TISO15693_SystemInfo* SystemInfo
    );
```

Parameters:	
`TISO15693_SystemInfo* SystemInfo`	Pointer to the structure which holds the System Information.
Return:	The return-value is the determined tag-type which is represented by one of the constants in the table below.

Definition	Value	Manufacturer	Tag Type
ISO15693_TAGTYPE_ICODESL2	0x00	NXP	ICode SL2
ISO15693_TAGTYPE_ICODESL2S	0x01		ICode SL2S
ISO15693_TAGTYPE_UNKNOWNNXP	0x0F		Unknown
ISO15693_TAGTYPE_TAGITHFIPLUSINLAY	0x10	TI	Tag-It HFI Plus Inlay
ISO15693_TAGTYPE_TAGITHFIPLUSCHIP	0x11		Tag-It HFI Plus Chip
ISO15693_TAGTYPE_TAGITHFISTD	0x12		Tag-It HFI Standard
ISO15693_TAGTYPE_TAGITHFIPRO	0x13		Tag-It HFI Pro
ISO15693_TAGTYPE_UNKNOWNTI	0x1F		Unknown
ISO15693_TAGTYPE_MB89R118	0x20	Fuji	MB89R118
ISO15693_TAGTYPE_MB89R119	0x21		MB89R119
ISO15693_TAGTYPE_MB89R112	0x22		MB89R112
ISO15693_TAGTYPE_UNKNOWNFUJI	0x2F		Unknown
ISO15693_TAGTYPE_24LR16	0x30	ST	24LR16
ISO15693_TAGTYPE_24LR64	0x31		24LR64
ISO15693_TAGTYPE_LRI1K	0x40		LRI1K
ISO15693_TAGTYPE_LRI2K	0x41		LRI2K
ISO15693_TAGTYPE_LRIS2K	0x42		LRIS2K
ISO15693_TAGTYPE_LRIS64K	0x43		LRIS64K
ISO15693_TAGTYPE_UNKNOWNST	0x4F		Unknown
ISO15693_TAGTYPE_SRF55V02P	0x50	Infineon	SRF55V02P
ISO15693_TAGTYPE_SRF55V10P	0x51		SRF55V10P
ISO15693_TAGTYPE_SRF55V02S	0x52		SRF55V02S
ISO15693_TAGTYPE_SRF55V10S	0x53		SRF55V10S
ISO15693_TAGTYPE_UNKNOWNINFINEON	0x5F		Unknown
ISO15693_TAGTYPE_UNKNOWN	0xFF	Unknown	Unknown ISO15693

Table 22.3: Retrievable tag types from System Information

22.3 Read/Write Data

22.3.1 Read Single Block

Read a single data block from the transponder. The function is available in two versions (Protocol Extension flag set or reset), as some tag types like ST 24LR16/64 require the Protocol Extension flag to be set for proper operation.

```
bool ISO15693_ReadSingleBlock
    (
    int BlockNumber,
    byte* BlockData,
    int* Length,
    int BufferSize
    );

    bool ISO15693_ReadSingleBlockExt
    (
    int BlockNumber,
    byte* BlockData,
    int* Length,
    int BufferSize
    );
```

Parameters:

`int BlockNumber`	This parameter holds the number of the block to be read.
`byte* BlockData`	This parameter holds the data which was read from the tag if the operation was successful. Note that the block size varies between different tag types, so the array size of `BlockData` should be set to a reasonable value.
`int* Length`	This parameter holds the length of data which was read from the tag in bytes.
`int BufferSize`	This parameter holds the array-size of `BlockData` in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

22.3.2 Write Single Block

Write to a single data block of the transponder. The function is available in two versions (Protocol Extension flag set or reset), as some tag types like ST 24LR16/64 require the Protocol Extension flag to be set for proper operation.

```
bool ISO15693_WriteSingleBlock
    (
    int BlockNumber,
    const byte* BlockData,
    int Length
    );

bool ISO15693_WriteSingleBlockExt
    (
    int BlockNumber,
    const byte* BlockData,
```

```
    int Length
    );
```

Parameters:

`int BlockNumber`	This parameter holds the number of the block to be written.
`const byte* BlockData`	This parameter holds the data which shall be written to the tag.
`int Length`	This parameter holds the length of data which shall be written to the tag in bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

23 LEGIC-Specific Functions

This chapter describes functions for accessing LEGIC functionality.

Notes:

- These functions are available at TWN4 MultiTech LEGIC only.
- The style of functions has been changed due to additional support of SM4500: All functions are starting with SM4X00 instead of SM4200. Old-style functions are supported via macros.

23.1 Direct Access of LEGIC Chip

TWN4 MultiTech LEGIC has a built-in LEGIC chip type SM4200 or SM4500. There are functions available to directly communicate with this chipset.

Note:

Due to license restrictions, this documentation only mentions the functions itself. In order to use full functionality of the LEGIC chip, appropriate documentation is required, which is available under NDA (none-disclosure agreement) only.

23.1.1 SM4X00_GenericRaw

Send a command and receive the response from SM4X00. Command and response are expected to include CRC. This function is intended to be used for end-to-end communication between SM4X00 and a host.

```
bool SM4X00_GenericRaw(const byte *TXData,int TXDataLength,
                       byte *RXData,int *RXDataLength,
                       int MaxRXDataLength,int Timeout);
```

Parameters:	
`const byte *TXData`	Pointer to an array of bytes, which contains the command to be sent to SM4X00.
`int TXDataLength`	Number of bytes to be sent to SM4X00.
`byte *RXData`	Pointer to an array of bytes, which receives response from SM4X00
`int *RXDataLength`	Pointer to an integer, which receives the actually read number of bytes.
`int MaxRXDataLength`	A value, which specifies the maximum number of bytes, which can be received byte `RXData`, thus the buffer size.
`int Timeout`	Maximum time, the function should wait for a response from SM4X00. This value is specified in milliseconds.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

23.1.2 SM4X00_Generic

Send a command and receive the response from SM4X00. This function is intended to be used by standand-along applications.

```
bool SM4X00_Generic(const byte *TXData,int TXDataLength,
                    byte *RXData,int *RXDataLength,
                    int MaxRXDataLength,int Timeout);
```

Parameters:	
`const byte *TXData`	Pointer to an array of bytes, which contains the command to be sent to SM4X00. The command has to be specified W/O leading length byte and W/O closing CRC value.
`int TXDataLength`	Number of bytes contained in `TXData`.
`byte *RXData`	Pointer to an array of bytes, which receives response from SM4X00. Received data is W/O length byte and W/O CRC value.
`int *RXDataLength`	Pointer to an integer, which receives length of the actually received payload.
`int MaxRXDataLength`	A value, which specifies the maximum number of bytes, which can be received byte `RXData`, thus the buffer size.
`int Timeout`	Maximum time, the function should wait for a response from SM4X00. This value is specified in milliseconds.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

23.1.3 SM4X00_StartBootloader

Start boot loader of SM4X00.

```
bool SM4X00_StartBootloader(byte *TLV,int *TLVLength,int MaxTLVLength)
```

Parameters:	
`byte *TLV`	
`int *TLVLength`	
`int MaxTLVLength`	
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

23.1.4 SM4X00_EraseFlash

Erase flash of SM4X00.

```
bool SM4X00_EraseFlash(void)
```

Parameters:	None.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

23.1.5 SM4X00_ProgramBlock

Program one block of data into the flash of SM4X00.

```
bool SM4X00_ProgramBlock(byte *Data,bool *Done)
```

Parameters:	
`byte *Data`	Pointer to an array of bytes.
`bool *Done`	Pointer to a boolean variable, which receives the status, if the last block was flashed.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

24 iCLASS Specific Transponder Operations

This chapter handles specific operations with iCLASS transponders. Prerequisites for this functionality are:

- The reader must be the TWN4 MultiTech/MultiTech Nano version, LEGIC is not supported.
- An iCLASS SIO card must be inserted into one of the SAM slots.
- The I-Option must be activated.
- For iCLASS Seos support, the SIO card must have firmware 1.19 or higher.

24.1 Read PAC Bits

This function can be used to read the PAC (Physical Access Control) bits from an iCLASS transponder. The transponder must have been selected before this function can be called.

```
bool ICLASS_GetPACBits
    (
    byte* PACBits,
    int* PACBitCnt,
    int MaxPACBytes
    );
```

Parameters:

`byte* PACBits`	After successful completion of this function, the buffer referred by this parameter holds the PAC bits read from the transponder. Take care for adequate dimensioning.
`int* PACBitCnt`	After successful completion of this function, this parameter holds the number of bits, the PAC contains.
`int MaxPACBytes`	This parameter holds the maximum number of bytes which the buffer `PACBits` can hold.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Remark: There are transponders available, that have not been configured to deliver the PAC bits. In this case, if an attempt is made to read these bits, the function returns `false`.

24.2 Example

The following example shows how to manually read the PAC from an iCLASS transponder using the built-in system functions.

```
byte ID[8];
int TagType;
int IDBitCnt;

byte PACBits[8];
int PACBitCnt;

// Search only for iCLASS transponders
SetTagTypes(0, TAGMASK(HFTAG_HIDICLASS));

while (true)
{
    // Search for transponders
    if (!SearchTag(&TagType,&IDBitCnt,ID,sizeof(ID)))
        continue;

    // Read the PAC bits
    if (!ICLASS_GetPACBits(PACBits, &PACBitCnt, sizeof(PACBits)))
        continue;

    // Output what was read from the card
    WriteHex(PACBits, PACBitCnt, (PACBitCnt+7)/8*2);
    WriteChar('\r');
}
```

25 FeliCa

This chapter handles specific operations of contactless transponders that support FeliCa technology. Before one of the following functions can be used, the transponder must have been selected using the function `SearchTag(...)`.

25.1 Polling

Use this function to acquire a card by specifying a System Code. The transponder only answers if the specified System Code matches to a system stored on the card. By specifying a wildcard (0xFF) for either the upper or lower byte, a particular match of System Code can be achieved.

```
bool FeliCa_Poll(uint16_t SystemCode, byte* IDm, byte* PMm);
```

Parameters:	
`uint16_t SystemCode`	Specify the two-byte System Code by this parameter.
`byte* IDm`	The Manufacture ID is returned by this buffer. The function always returns 8 bytes.
`byte* PMm`	The Manufacture Parameter is returned by this buffer. The function always returns 8 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

25.2 Request System Code

Use this function to aquire a list of System Codes which are available on the card. This function does not work with FeliCa Lite or FeliCa Plug ICs.

```
bool FeliCa_RequestSystemCode
    (
    int* NumberOfSystemCodes,
    uint16_t* SystemCodeList,
    int MaxNumberOfSystemCodes
    );
```

Parameters:	
`int* NumberOfSystemCodes`	This parameter holds the number of retrieved System Codes.
`uint16_t* SystemCodeList`	This parameter holds the list of System Codes which are available on the card.
`int MaxNumberOfSystemCodes`	Specify the maximum number of System Codes, that can be stored in the array `SystemCodeList`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

25.3 Request Service

Use this function to verify the existance of Area and Service Codes. The function returns the Key Version of existing Area and System Codes. If the specified Area or System does not exist, the respective Key Version is 0xFFFF. This function does not work with FeliCa Lite or FeliCa Plug ICs.

```
bool FeliCa_RequestService
    (
    int NumberOfServices,
    const uint16_t* ServiceCodeList,
    uint16_t* KeyVersionList
    );
```

Parameters:	
`int NumberOfServices`	This parameter specifies the size of `ServiceCodeList`.
`const uint16_t* ServiceCodeList`	This array holds the list of Service Codes that shall be queried.
`uint16_t* KeyVersionList`	The queried KeyVersions are returned by this array. It has the same size as `ServiceCodeList`, each KeyVersion is assigned to the order of appearance of `ServiceCodeList`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

25.4 Read Without Encryption

Use this function to read blocks of data from a authentication-not-required service. This function works with all transponders supporting FeliCa technology.

```
bool FeliCa_ReadWithoutEncryption
    (
    int NumberOfServices,
    const uint16_t* ServiceCodeList,
    int NumberOfBlocks,
    const uint16_t* BlockList,
    byte* Data
    );
```

Parameters:

`int NumberOfServices`	This parameter specifies the size of `ServiceCodeList`.
`const uint16_t* ServiceCodeList`	This array holds the list of Service Codes. Currently, one Service Code can be specified.
`int NumberOfBlocks`	This parameter specifies the number of blocks that shall be read. The current implementation allows reading of four blocks at a time.
`const uint16_t* BlockList`	This array holds the list of blocks that shall be read.
`byte* Data`	Block data which was read from the card is returned by this buffer. A block has always 16 bytes of data, so the buffer must be dimensioned depending on the number of blocks that shall be read. The block data is returned in the order of appearance of the values of `BlockList`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

25.5 Write Without Encryption

Use this function to write blocks of data to a authentication-not-required service. This function works with all transponders supporting FeliCa technology.

```
bool FeliCa_WriteWithoutEncryption
    (
    int NumberOfServices,
    const uint16_t* ServiceCodeList,
    int NumberOfBlocks,
    const uint16_t* BlockList,
    const byte* Data
    );
```

Parameters:

`int NumberOfServices`	This parameter specifies the size of `ServiceCodeList`.
`const uint16_t* ServiceCodeList`	This array holds the list of Service Codes. Currently, one Service Code can be specified.
`int NumberOfBlocks`	This parameter specifies the number of blocks that shall be written. The current implementation allows writing of four blocks at a time.
`const uint16_t* BlockList`	This array holds the list of blocks that shall be written.
`const byte* Data`	Block data which shall be written to the card. A block has always 16 bytes of data, so the buffer must hold `NumberOfBlocks` multiplied by 16 bytes of data. The block data must be arranged in the order of appearance of the values of `BlockList`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

25.6 Transparent Data Exchange

This function can be used for transparent exchange of data between reader and FeliCa transponders, e.g. for transponder commands which are not covered by the current implementation of the reader API.

```
bool FeliCa_TDX
    (
    byte* TXRX,
    int TXByteCnt,
    int* RXByteCnt,
    int MaxRXByteCnt,
    byte MaximumResponseTime,
    byte NumberOfBlocks
    );
```

Parameters:

`byte* TXRX`	This buffer holds the byte-string that shall be transmitted to the transponder. The response of the transponder is also returned by this parameter. Take care for adequate dimensioning.
`int TXByteCnt`	This parameter holds the number of bytes that shall be transmitted to the transponder.
`int* RXByteCnt`	After successful completion of this function, this parameter holds the number of bytes that the transponder response contains.
`int MaxRXByteCnt`	This parameter holds the array-size of `TXRX` in bytes.
`byte MaximumResponseTime`	This parameter holds the parameter byte which shall be used for calculation of the Maximum Response Time according to the calculation formula.
`byte NumberOfBlocks`	This parameter holds the value n which shall be used for calculation of the Maximum Response Time according to the calculation formula.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

26 Topaz Specific Transponder Operations

26.1 Read UID

Use this function to manually read byte 0 to 3 of the UID and the header ROM. The remaining bytes 4 to 6 of the UID can be read e.g. by using the function `Topaz_ReadByte`.

```
bool Topaz_RID(byte* HR0, byte* HR1, byte* UID);
```

Parameters:	
`byte* HR0`	The byte HR0 of the Header ROM is returned by this parameter.
`byte* HR1`	The byte HR1 of the Header ROM is returned by this parameter.
`byte* UID`	The UID bytes 0 to 3 are returned by this buffer.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

26.2 Read Data

In order to read data from memory of Topaz transponders, there are two functions available. You may choose between reading one single byte or even read the entire memory space.

26.2.1 Read Single Byte

Use this function to read one single byte from the memory of the transponder.

```
bool Topaz_ReadByte(const byte* UID, byte ADD, byte* Byte);
```

Parameters:	
`const byte* UID`	Specify byte 0 to 3 of the UID by this parameter.
`byte ADD`	Specify the address in memory to be read from.
`byte* Byte`	This parameter holds the byte which was read from the transponder.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

26.2.2 Read All Transponder Data

Use this function to read the entire memory of the transponder.

```
bool Topaz_ReadAllBlocks(const byte* UID, byte* HR0, byte* HR1, byte* Data);
```

Parameters:	
`const byte* UID`	Specify byte 0 to 3 of the UID by this parameter.
`byte* HR0`	The byte HR0 of the Header ROM is returned by this parameter.
`byte* HR1`	The byte HR1 of the Header ROM is returned by this parameter.
`byte* Data`	The transponder memory data is returned by this buffer. The function returns 120 bytes, so take care for proper dimensioning.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

26.3 Write Data

In order to write data to the memory of Topaz transponders, there are two functions available. You may chose between programming of data with preceding erase or without erase. So if the variant without erase is selected, this results in logical ORing of data bits. In the initial state, all writeable data bytes of Topaz are 0x00.

26.3.1 Write Single Byte With Erase

Use this function to write one byte to the memory of the transponder. A preceding erase cycle is performed prior programming takes place.

```
bool Topaz_WriteByteWithErase(const byte* UID, byte ADD, byte Byte);
```

Parameters:	
`const byte* UID`	Specify byte 0 to 3 of the UID by this parameter.
`byte ADD`	Specify the address in memory to be written to.
`byte Byte`	This parameter holds the byte to be written.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

26.3.2 Write Single Byte Without Erase

Use this function to write one byte to the memory of the transponder. As no preceding erase cycle is performed prior programming, the overall operation results in logical ORing of the existing data byte on the transponder and the byte to be written.

```
bool Topaz_WriteByteNoErase(const byte* UID, byte ADD, byte Byte);
```

Parameters:	
`const byte* UID`	Specify byte 0 to 3 of the UID by this parameter.
`byte ADD`	Specify the address in memory to be written to.
`byte Byte`	This parameter holds the byte to be written.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

27 Simple NDEF Exchange Protocol (SNEP)

This chapter handles transmission of NDEF (NFC Data Exchange Format) messages between a TWN4 reader and a NFC enabled device using the Simple NDEF Exchange Protocol. For message exchange, a NFC Peer-to-Peer connection must have been established.
The SNEP service provides both a logical In-Box and a logical Out-Box for receiving and transmitting messages. Each message box works as FIFO, which enables reader and host-software to exchange even large messages as a stream of data. This functionality is also useful to reduce outbound buffering on host side. Each message box can manage only one message at the same time, so message-queuing is currently not supported. Large messages that do not fit into the internal FIFO must be transmitted fragmented, so the sending side must break up the message into smaller parts that fit into the FIFO, the receiving side must reassemble the parts as a consequence. When dealing with large messages, it might become necessary to read data from the FIFO fast enough during a ongoing transmission in order to prevent any tailbacks.
Note: This functionality is only available on TWN4 MultiTech based on Core Module.

27.1 Initialize SNEP Service

Use this function for initialization and starting of the built-in SNEP service. The function should be called at last once before issuing `SearchTag()` with TagType `HFTAG_NFCP2P` enabled. Depending on the implementation of the counterpart NFC device, there might be a delay until the SNEP service is activated on both communication peers. This time usually ranges around 100 ms up to 500 ms.

```
bool SNEP_Init(void);
```

Parameters:	None.
Return:	If the SNEP service was successfully started, the return value is `true`, otherwise it is `false`.

27.2 Get Connection State

Use this function to query the current connection state of the SNEP service. This can be used for checking e.g. any loss of the physical NFC Peer-to-Peer connection.

```
int SNEP_GetConnectionState(void);
```

Parameters:	None.
Return:	SNEP_STATE_DEINIT: The SNEP service has not been started.
	SNEP_STATE_SLEEP: The SNEP service has been started, but there is no active connection.
	SNEP_STATE_IDLE: The SNEP service is running, but there is currently no active exchange of messages.
	SNEP_STATE_CONNCLIENT: The SNEP service is running in client mode.
	SNEP_STATE_CONNSERVER: The SNEP service is running in server mode.

27.3 Query Message FIFO

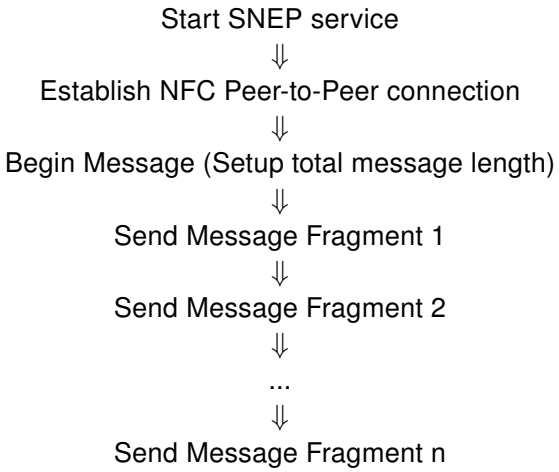
Use this function to get information of the respective message FIFO.

```
int SNEP_GetFragmentByteCount(int Direction);
```

Parameters:	
int Direction	Specify the message box to be queried by this parameter. Valid values are DIR_OUT (Out-Box) or DIR_IN (In-Box), use one of these predefined constants.
Return:	If the In-Box is queried, the return value is the current number of bytes which are available for reading from host side. If the Out-Box is queried, the return value is the number of bytes that can be written to the FIFO.

27.4 Transmit NDEF Message

This section handles transmission of NDEF messages. A typical communication flow for transmitting a NDEF message looks like this:



27.4.1 Begin Message

Use this function to setup the total message length. A message can reach up to 4 GBytes.

```
bool SNEP_BeginMessage(uint32_t MsgByteCnt);
```

Parameters:	
`uint32_t MsgByteCnt`	Specify the total message length by this parameter.
Return:	If the operation was successful, the return value is `true`. If a previously set up message has not been transmitted completely, the return value is `false`.

27.4.2 Send Message Fragment

Use this function to store a fragment of a message in the Out-Box FIFO. The message must be transmitted completely in order to make the FIFO available for new outgoing messages.

```
bool SNEP_SendMessageFragment(const byte* MsgFrag, int FragByteCnt);
```

Parameters:	
`const byte* MsgFrag`	Specify the buffer that holds the message fragment by this parameter.
`int FragByteCnt`	This parameter holds the length the message fragment.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

27.4.3 Example

The following example demonstrates transmission of a NDEF message from a TWN4 MultiTech reader to another NFC device running SNEP:

```
const byte NDEF_Message[] = { /* Your NDEF message */ };

void TransmitNDEFMessage(void)
{
    // Wait for SNEP service is running
    unsigned long SNEPConnectionStartTime = GetSysTicks();
    // SNEP service must be at least in IDLE state
    while (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
    {
        if (GetSysTicks() - SNEPConnectionStartTime > 500)
            return;
    }
    // Transmit NDEF message as long as a NFC connection is established
    int FragmentOffset = 0;
    int NDEF_MessageByteCnt = sizeof(NDEF_Message);
    while (true)
    {
        if (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
            return;
        // Get available buffer size from operating system for message fragmenting
        FragmentSize = SNEP_GetFragmentByteCount(DIR_OUT);
        if (FragmentSize > 0)
        {
            // Is this the first fragment?
            if (FragmentOffset == 0)
            {
                // Yes, Setup message
                if (!SNEP_BeginMessage(NDEF_MessageByteCnt))
                    return;
            }
            // Calculate fragment size
            if (NDEF_MessageByteCnt - FragmentOffset <= FragmentSize)
                FragmentSize = NDEF_MessageByteCnt - FragmentOffset;
            // Send a fragment of the message
            if (!SNEP_SendMessageFragment(&NDEF_Message[FragmentOffset], FragmentSize))
                return;
            FragmentOffset += FragmentSize;
        }
        // Was the message completely tansmitted?
        if (FragmentOffset == NDEF_MessageByteCnt)
            return;
    }
}

#define MAXIDBYTES  10

byte ID[MAXIDBYTES];
int IDBitCnt;
int TagType;

int main(void)
{
    // Enable NFC Peer-to-Peer mode
```

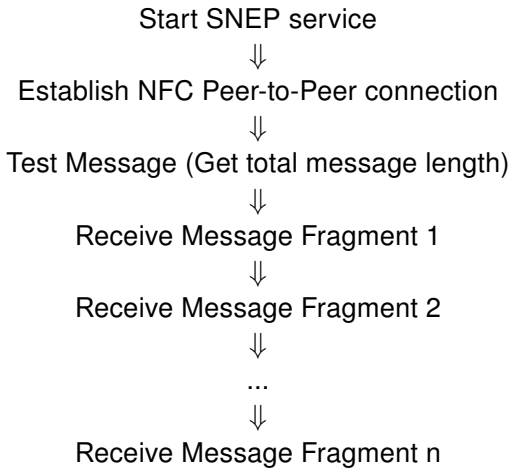
```
    SetTagTypes(0, TAGMASK(HFTAG_NFCP2P));

    // Start SNEP service
    SNEP_Init();

    while (true)
    {
        // Search a transponder
        if (SearchTag(&TagType, &IDBitCnt, ID, sizeof(ID)))
        {
            if (TagType == HFTAG_NFCP2P)
            {
                // Transmit NDEF message
                TransmitNDEFMessage();
            }
        }
    }
}
```

27.5 Receive NDEF Message

This section handles reception of NDEF messages. A typical communication flow for receiving a NDEF message looks like this:



27.5.1 Test Message

Use this function to test if there is a new message available in the In-Box. The function returns the total length of the message. A message can reach up to 4 GBytes.

```
bool SNEP_TestMessage(uint32_t* MsgByteCnt);
```

Parameters:	
`uint32_t* MsgByteCnt`	The total message length is returned by this parameter.
Return:	If a message is available, the return value is `true`, otherwise it is `false`.

27.5.2 Receive Message Fragment

Use this function to receive a fragment of a message stored in the In-Box FIFO. A message must be read completely from the FIFO in order to make it available for new incoming messages.

```
bool SNEP_ReceiveMessageFragment(byte* MsgFrag, int FragByteCnt);
```

Parameters:	
`byte* MsgFrag`	Specify the buffer that holds the message fragment by this parameter.
`int FragByteCnt`	This parameter holds the length the message fragment to be read.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

27.5.3 Example

The following example demonstrates reception of a NDEF message from another NFC device running SNEP:

```
void ReceiveNDEFMessage(void)
{
    // Wait for SNEP service is running
    unsigned long SNEPConnectionStartTime = GetSysTicks();
    // SNEP service must be at least in IDLE state
    while (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
    {
        if (GetSysTicks() - SNEPConnectionStartTime > 500)
            return;
    }
    // Receive all NDEF messages as long as a NFC connection is established
    while (true)
    {
        uint32_t MessageSize;
        byte Message[4096];
        // Wait for a incoming NDEF message or loss of connection
        while (!SNEP_TestMessage(&MessageSize))
        {
            if (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
                return;
        }
        // A NDEF message was announced. Now read it.
        int FragmentOffset, FragmentSize;
        for (FragmentOffset = 0; FragmentOffset < MessageSize; FragmentOffset += FragmentSize)
        {
        // Wait, till fragment of the message arrives
            do
            {
                if (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
                    return;
                FragmentSize = SNEP_GetFragmentByteCount(DIR_IN);
            }
            while (FragmentSize == 0);
            SNEP_ReceiveMessageFragment(&Message[FragmentOffset],FragmentSize);
        }
        // We read the entire NDEF message
    }
}

#define MAXIDBYTES  10

byte ID[MAXIDBYTES];
int IDBitCnt;
int TagType;

int main(void)
{
    // Enable NFC Peer-to-Peer mode
    SetTagTypes(0, TAGMASK(HFTAG_NFCP2P));

    // Start SNEP service
    SNEP_Init();
```

```
    while (true)
    {
        // Search a transponder
        if (SearchTag(&TagType, &IDBitCnt, ID, sizeof(ID)))
        {
            if (TagType == HFTAG_NFCP2P)
            {
                // Receive NDEF message
                ReceiveNDEFMessage();
                DoSomething();
            }
        }
    }
}
```

28 BLE Functions

The reader TWN4 MultiTech 2 / 3 BLE supports LF / HF transponders and additionally BLE (Bluetooth Low Energy). This allows to connect to all devices with the Bluetooth Standard 4.0 or greater: Android mobile phones with Version 4.3 or greater, iPhones 4S, 5 or greater and PCs with Windows 8.1 / 10 and Bluetooth hardware.

The App in the TWN4 MultiTech 2 / 3 BLE control the BLE module. There are commands for initialization, setting connection parameters, do connection and f.e. reading / writing GATT data fields.

First of all initialize the BT Module. To make the extensive setting easier, simply call the function `BLEInit` to set the wished configuration for starting the Module. The Mode parameter fills the environment variables for the selected mode.

To set an own environment, use the functions `BLEPresetConfig` and `BLEPresetUserData` followed with `BLEInit(0)`.

After initialization call the function `BLECheckEvent` for checking events of Bluetooth. It's good to use a call frequency of about 100ms. This would be fine. Slower calling slows the BLE functionality. Faster is not necessary but no problem.

Environmet information are called by `BLEGetAddress` for the address of the reader, the address for the connected device and the type of this address. Information of the firmware ask with `BLEGetVersion` and at least connection environment with `BLEGetEnvironment`.

The GATT (Generic Attribute Profile) on the BLE module is reading with `BLEGetGattServerAttribute-Value` and writing with `BLESetGattServerAttributeValue`.

To request the latest RSSI call the function `BLERequestRssi` if a connection has established. The RSSI value is returned by the event BLE_EVENT_CONNECTION_RSSI. Closing a connection is thrown with `BLERequestEndpointClose`. But also an automatic closing is carried out by the set timeout at initialization.

The BLE Module on the TWN4 MultiTech 2 BLE communicates serial with the main core. COM2 is reserverd for the communication with the BLE Modul and GPIO7 is the reset of the BLE Module. So do not use COM2 and GPIO7 for other things on the hardware TWN4 MultiTech 2 / 3 BLE.

28.1 BLEPresetConfig

This function presets the individual configuration structure for the BLE module. The initialization command `BLEInit` is necessary after this - optional also the `BLEPresetUserData`.

```
bool BLEPresetConfig
        (
        TBLEConfig* BLEConfig
        );
```

Parameters:

`TBLEConfig*` Reference to the structure that holds the BLE configuration parameters. See the description of `TBLEConfig` for details.

Return: If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
uint32_t ConnectTimeout	32	Timout of an established connection in milliseconds.
`byte Power`	8	TX power in 0.1dBm steps in the range 0 to 80 (0.0dBm to 8.0dBm).
`byte BondableMode`	8	Bonding: 0 = Off, 1 = On. If additionaly Bit7=1, then self defined UserData (with function BLEPresetUserData(..)) are used.
`uint16_t AdvInterval`	16	Advertisement interval: values 20ms to 10240ms
`byte ChannelMap`	8	Advertisement Bluetooth channels: 1 = CH37, 2 = CH38, 3 = CH37 + CH38, 4 = CH39, 5 = CH37 + CH39, 6 = CH38 + CH39, 7 = CH37 + CH38 + CH39.
`byte DiscoverMode`	8	Discoverable Modes for the LE (Low Energy) GAP (Generic Access Profile): 0 = non discoverable, 1 = limited discoverable, 2 = general discoverable, 3 = braodcast, 4=user data.
`byte ConnectMode`	8	Connectable Modes for the LE (Low Energy) GAP (Generic Access Profile): 0 = non connectable, 1 = directed connectable, 2 = undirected connectable, 3 = scannable / non connectable.
`byte SecurityFlags`	8	Security requirement bitmask: Bit 0 = 0: Allow bonding without MITM protection, Bit 0 = 1: Bonding requires MITM protection, Bit 1 = 0: Allow encryption without bonding, Bit 1 = 1: Encryption requires bonding, Bit 2 to 7: Reserved, Default value: 0x00.

byte IOCapabilities	8	Security Management related I/O capabilities: 0 = display only, 1 = display yes/no, 2 = keyboard only, 3 = no input / no output, 4 = keyboard / display
uint32_t Passkey	32	Passkey: If security is configured, the application needs to display or ask user to enter a passkey during the bonding process. See: BLE_EVENT_SM_PASSKEY_DISPLAY or BLE_EVENT_SM_PASSKEY_REQUEST.

Table 28.1: Definition of TBLEConfig

28.2 BLEPresetUserData

F.e. the Apple company has introduced iBeacons to broadcast their identifier to nearby portable electronic devices. If you discover an iBeacon or common a Beacon, you get his UUID, Major and Minor values. With the TWN4 MultiTech 2 BLE, you can configure the reader to be a Beacon.

```
bool BLEPresetUserData
        (
        byte ScanResp,
        const byte* UserData,
        int UserDataLength
        );
```

Parameters:

byte ScanResp	Selection the type showing user data: 0 = advertise packets, 1 = scan packets.
const byte* UserData	Reference to the byte buffer that holds the UserData parameters. See the description of UserData for details.
int UserDataLength	Length of the UserData. Maximum data length is 30 Bytes.
Return:	If the operation was successful, the return value is true, otherwise it is false.

Members	Length (Bits)	Value (f.e.)	Description
UserData[0]	8	0x02	Length of the Flags field - 2 bytes.
UserData[1]	8	0x01	Length of the Flags field - high byte.
UserData[2]	8	0x06	Length of the Flags field - low byte.
UserData[3]	8	0x1A	Length of the Manufacturer Data field - 26 bytes.
UserData[4]	8	0xFF	Data type / Manufacturer specific data / Type of the Manufacturer Data field.

UserData[5]	8	0x4C	Manufacturer data - high byte, Company ID field - 0x4C00 = Apple's Bluetooth SIG ID.
UserData[6]	8	0x00	Manufacturer data - low byte.
UserData[7]	8	0x02	Manufacturer data - high byte, Beacon Type field - 0x0215 = iBeacon.
UserData[8]	8	0x15	Manufacturer data - low byte.
UserData[9]	8	0xE2	UUID E2C56DB5-DFFB-48D2-B060-D0F5A71096-E0 (Apple AirLocate Service)
UserData[10]	8	0xC5	
UserData[11]	8	0x6D	
UserData[12]	8	0xB5	
UserData[13]	8	0xDF	
UserData[14]	8	0xFB	
UserData[15]	8	0x48	
UserData[16]	8	0xD2	
UserData[17]	8	0xB0	
UserData[18]	8	0x60	
UserData[19]	8	0xD0	
UserData[20]	8	0xF5	
UserData[21]	8	0xA7	
UserData[22]	8	0x10	
UserData[23]	8	0x96	
UserData[24]	8	0xE0	
UserData[25]	8	0x00	The Major high value, which is used to group related beacons.
UserData[26]	8	0x00	The Major low value.
UserData[27]	8	0x00	The Minor high value, which is used to specify individual beacon within a group.
UserData[28]	8	0x00	The Minor low value.
UserData[29]	8	0xC3	Signal power (calibrated RSSI) - See the iBeacon specification for measurement guidelines.

Table 28.2: Definition of UserData

28.3 BLEInit

This function initialize the Bluetooth BLE Module on the reader. Different modes are possible: The custom mode makes individual operating modes possible - pre configured with BLEPresetConfig and BLEPresetUserData. The other modes are predefined modes for advertisment and Beacon.

```
bool BLEInit
```

```
(
int Mode
);
```

Parameters:

`int Mode`	Specify the initialization mode. See the definition of `Mode` for meaning of each member.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

BLE_MODE_CUSTOM	0	BLE Custom mode for previously defined configuration with functions `BLEPresetConfig` and `BLEPresetUserData`.
BLE_MODE_ADVERTISEMENT	1	Easy BLE advertisement mode with no encryption and no bonding.
BLE_MODE_BEACON	2	BLE Beacon mode for mobile devices.
BLE_MODE_ADVERTISEMENT_SM	3	BLE advertisement mode with Security Management: Bonding with no input and no output (SecurityFlags=3).
BLE_MODE_OFF	255	BLE is no longer active.

Table 28.3: Definition of `Mode`

28.4 BLECheckEvent

This function returns the actual event of the BLE module. The returned event tells different status messages of the Bluetooth environment either information or user action.

```
int BLECheckEvent
        (
        void
        );
```

Parameters:	None.
Return:	Specify the event. See the definition in the table below

BLE_EVENT_NONE	0x00	No event.
BLE_EVENT_ENDPOINT_CLOSING	0x21	This event indicates that an endpoint is closing or that the remote end has terminated the connection.
BLE_EVENT_ENDPOINT_DATA	0x22	This event indicates incoming data from an endpoint.
BLE_EVENT_ENDPOINT_STATUS	0x23	This event indicates an endpoint's status.
BLE_EVENT_ENDPOINT_SYNTAX_ERROR	0x24	This event indicates that a protocol error to the BLE module was detected.
BLE_EVENT_GATT_MTU_EXCHANGED	0x45	This event indicates that a GATT MTU exchange procedure has been completed.

BLE_EVENT_GATT_SERVER_ATTRIBUTE _VALUE	0x51	This event indicates that the value of an attribute in the local GATT database has been changed by a remote GATT client.
BLE_EVENT_GATT_SERVER_CHARACTERISTIC _STATUS	0x52	This event indicates either that a local Client Characteristic Configuration descriptor has been changed by the remote GATT client, or that a confirmation from the remote GATT client was received upon a successful reception of the indication.
BLE_EVENT_CONNECTION_CLOSED	0x71	This event indicates that a connection was closed.
BLE_EVENT_CONNECTION_OPENED	0x72	This event indicates that a new connection was opened, whether the devices are already bonded, and what is the role of the Bluetooth device (Slave or Master). An open connection can be closed with the command BLERequestEndpointClose.
BLE_EVENT_CONNECTION_PARAMETERS	0x73	This event is triggered whenever the connection parameters are changed and at any time a connection is established.
BLE_EVENT_CONNECTION_RSSI	0x74	This event is triggered when an BLERequestRssi command has completed.
BLE_EVENT_LE_GAP_SCAN_RESPONSE	0x81	This event reports any advertisement or scan response packet that is received by the device's radio while in scanning mode.
BLE_EVENT_SM_BONDED	0x91	This event is triggered after the pairing or bonding procedure has been successfully completed.
BLE_EVENT_SM_BONDING_FAILED	0x92	This event is triggered if the pairing or bonding procedure has failed.
BLE_EVENT_SM_LIST_ALL_BONDINGS _COMPLETE	0x93	This event is triggered by the BLE_EVENT_SM_LIST_BONDING _ENTRY.
BLE_EVENT_SM_LIST_BONDING_ENTRY	0x94	This event is triggered if bondings exist in the local database.
BLE_EVENT_SM_PASSKEY_DISPLAY	0x95	This event indicates a request to display the passkey to the user.
BLE_EVENT_SM_PASSKEY_REQUEST	0x96	This event indicates a request for the user to enter the passkey displayed on the remote device.
BLE_EVENT_SM_CONFIRM_BONDING	0x99	This event indicates a request to display that new bonding request has been received to the user and for the user to confirm the request.

BLE_EVENT_SM_CONFIRM_PASSKEY	0x9A	This event indicates a request to display the passkey to the user and for the user to confirm the displayed passkey.
BLE_EVENT_SYSTEM_BOOT	0xA2	This event indicates the device has started and the radio is ready. This even carries the firmware build number and other SW and HW identification codes saved in the function BLEGetVersion.

Table 28.4: Definition of Event

28.5 BLEGetAddress

This function returns the device address from the BLE module, the remote address from the connected device and the address type of the remote address.

```
bool BLEGetAddress
        (
        byte* DeviceAddress,
        byte* RemoteAddress,
        byte* Type
        );
```

Parameters:

byte* DeviceAddress	The device address of the BLE module in 6 bytes hex is returned by this parameter.
byte* RemoteAddress	The remote address of the connected device in 6 bytes hex is returned, if the remote device is successfull connected. For additional information of the remote address see the Type parameter.
byte* Type	The type of the remote address is returned by this parameter. Possible values are: 0 = public address, 1 = random address, 2 = public identity address resolved by stack, 3 = random identity address resolved by stack, 4 = Classic Bluetooth address.
Return:	If the operation was successful, the return value is true, otherwise it is false.

28.6 BLEGetVersion

This function returns on the one hand the version string of the BLE module firmware in ASCII format and on the other the boot string of the BLE hardware.

```
bool BLEGetVersion
        (
        byte* HWVersion,
        byte* BootString
```

```
        );
```

Parameters:

`byte* HWVersion`	The firmware version string (16 bytes) in ASCII code is returned by this parameter. Example: "V1.03,14.11.2016"
`byte* BootString`	The boot string of the BLE hardware is returned. The information is binary coded in 12 bytes with the following information: Byte 0 - 1: Major release version, Byte 2 - 3: Minor release version, Byte 4 - 5: Patch release number, Byte 6 - 7: Build number, Byte 8 - 9: Bootloader version, Byte 10 - 11: Hardware type.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.7 BLEGetEnvironment

This function can be used to ask for connection enviroment of a connected device.

```
bool BLEGetEnvironment
        (
        byte* DeviceRole,
        byte* SecurityMode,
        byte* Rssi
        );
```

Parameters:

`byte* DeviceRole`	The device role of the connection is returned: 0 = Slave, 1 = Master.
`byte* SecurityMode`	The security mode of the established connection is returned. Possible values are: 0 = No security (mode 1 level1) 1 = Unauthenticated pairing with encryption (mode 1 level 2) 2 = Authenticated pairing with encryption (mode 1 level 3).
`byte* Rssi`	RSSI of the BLE connection Range: -127 to +20. Units: dBm.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.8 BLEGetGattServerAttributeValue

This function returns the data of a GATT attribute handle.

```
bool BLEGetGattServerAttributeValue
        (
        int AttrHandle,
```

```
        byte *Data,
        int *Len,
        int MaxLen
        );
```

Parameters:

`int AttrHandle`	Specify the attribute handle number of the GATT who is selected to read. See the GATT table for possible values.
`byte *Data`	The read data of the given attribute handle is returned by this parameter.
`int *Len`	This parameter holds the length of data which was read from the GATT.
`int MaxLen`	Maximum number of characters, the specified byte array can receive excluding the 0-termination.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.9 BLESetGattServerAttributeValue

This function writes data to an attribute handle. Notice that the GATT attribute must be writeable.

```
bool BLESetGattServerAttributeValue
        (
        int AttrHandle,
        int Offset,
        const byte *Data,
        int Len
        );
```

Parameters:

`int AttrHandle`	Specify the attribute handle number of the GATT for writing data. For possible values see the GATT table.
`int Offset`	Specify the starting address for writing to data. The valid range of this parameter is 0 to Len-1.
`byte *Data`	The write data buffer to the attribute handle with the specifyed offset.
`int Len`	This parameter holds the length of data which shall be written to the GATT.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.10 BLERequestRssi

This function calls a request for the actual RSSI. The value of the RSSI is returned by an event `BLE_EVENT_CONNECTION_RSSI` with function `BLECheckEvent`. The function makes only sense if there is a established connection with a remote device.

```
        bool BLERequestRssi
        (
        void
        );
```

Parameters:	None.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.11 BLERequestEndpointClose

This function closes a connection with the remote device. If the connection is closed, the function `BLECheckEvent` returns the event `BLE_EVENT_CONNECTION_CLOSED` for successfull closing.

```
bool BLERequestEndpointClose
        (
        void
        );
```

Parameters:	None.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.12 BLEGetGattServerCharacteristicStatus

This function can be used after the event BLE_EVENT_GATT_SERVER_CHARACTERISTIC_STATUS to ask for GATT field characteristic change through a client. With the same function ask after the event BLE_EVENT_GATT_SERVER_ATTRIBUTE_VALUE for the GATT AttrHandle number.

```
bool BLEGetGattServerCharacteristicStatus
        (
        int *AttrHandle,
        int *AttrStatusFlag,
        int *AttrConfigFlag
        );
```

Parameters:

`int *AttrHandle`	GATT characteristic handle.
`int *AttrStatusFlag`	Describes whether Client Characteristic Configuration was changed or if confirmation was received: 1 = Characteristic client configuration has been changed. 2 = Characteristic confirmation has been received.
`int *AttrConfigFlag`	This field carries the new value of the Client Characteristic Configuration: 0 = Disable notifications and indications. 1 = Notification. 2 = Indication.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

28.13 BLEFindGattServerAttribute

This command can be used to find attributes of certain type from a local GATT database. Type is usually given as 16-bit (2 byte) or 128-bit (16 byte) UUID.

```
bool BLEFindGattServerAttribute
(
        const byte *UUID,
        int UUIDLength,
        int *AttrHandle
);
```

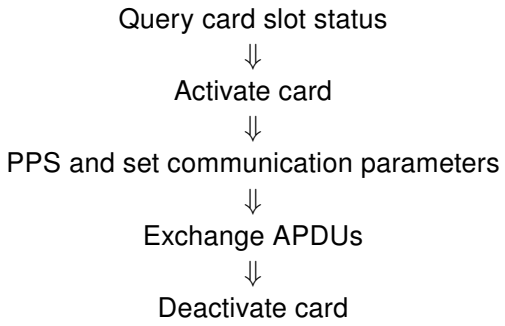
Parameters:

`const byte *UUID`	16-bit or 128-bit UUID of the local GATT database.
`int UUIDLength`	Length of the UUID in bytes (e.g. 2 or 16).
`int *AttrHandle`	The GATT characteristic attribute handle is returned by this parameter.
Return:	If the operation was successful, the return value is `true`, otherwise it is returned `false`.

29 Contact Card Operations

29.1 Microprocessor Cards

This chapter handles the usage of ISO7816 compliant Integrated Circuit Cards such as ID-1 or SAM (Secure Access Module) cards. The TWN4 ISO7816 API offers different system functions for covering different imaginable scenarios. A typical communication flow with contact cards looks like this:



29.1.1 Query Card Slot Status

This function shall be used to query information of the physical card slot status, e.g. to find out if a card is inserted or not. The function returns the slot status in CCID compliant style, this means it return information about slot status, error information and clock status. The internal state of the card is not changed. Please note, depending on the used hardware (TWN4 Desktop or TWN4 SmartCard) the amount of retrievable information differs.

```
bool ISO7816_GetSlotStatus(int Channel, TISO7816SlotStatus* SlotStatus);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`TISO7816SlotStatus* SlotStatus`	The card slot status is returned by this parameter. See the definition of `TISO7816SlotStatus` for meaning of each member.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
TISO7816StatusReg bStatus	8	Slot status register compliant to CCID. See the definition of TISO7816StatusReg for meaning of the different bit fields.
byte bError	8	Error code compliant to CCID.
byte bClockStatus	8	Clock status information compliant to CCID. Possible values are: ISO7816_CLOCKSTATUS_RUNNING, ISO7816_CLOCKSTATUS_CLKSTPL, ISO7816_CLOCKSTATUS_CLKSTPH, ISO7816_CLOCKSTATUS_CLKSTPU.

Table 29.1: Definition of TISO7816SlotStatus

Members	Length (Bits)	Description
byte bmICCStatus	2	Physical status of the card slot. Possible values are: ISO7816_ICCPRESENTANDACTIVE, ISO7816_ICCPRESENTANDINACTIVE and ISO7816_NOICCPRESENT.
byte bmRFU	4	These bits are reserved for future use.
byte bmCommandStatus	2	Command status information compliant to CCID.

Table 29.2: Definition of TISO7816StatusReg

29.1.2 Card Activation

This function shall be used to activate and initialize communication with the card inserted in one of the slots connected to the TWN4 reader. All communication parameters are reset to default. Depending on the hardware platform, the reader shows different behaviour regarding reset-handling of the card: On TWN4 Desktop, calling this function always leads to a warm reset, on TWN4 SmartCard, the first call performs a cold reset and any subsequent function call leads to a warm reset until the card is deactivated. The result of the entire operation is the receipt of the Answer To Reset (ATR) from the card. Based on the content of the ATR, the user may decide how to further proceed with the card. Note that selection of voltage level is only available for TWN4 SmartCard.

```
bool ISO7816_IccPowerOn
    (
    int Channel,
    byte* ATR,
    int* ATRByteCnt,
    int MaxATRByteCnt,
    byte bPowerSelect,
    TISO7816StatusReg* bStatus,
    byte* bError
    );
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`byte* ATR`	After successful completion of this function, the buffer referred by this parameter holds the ATR which was read from the card. Take care for adequate dimensioning.
`int* ATRByteCnt`	After successful completion of this function, this parameter holds the number of bytes, the ATR contains.
`int MaxATRByteCnt`	This parameter holds the array-size of `ATR` in bytes.
`byte bPowerSelect`	Specify the operating voltage level which shall be used for the card. Valid values are `ISO7816_POWERSELECT_AUTO`, `ISO7816_POWERSELECT_5V`, `ISO7816_POWERSELECT_3V`, or `ISO7816_POWERSELECT_1V8`, use one of these predefined constants.
`TISO7816StatusReg* bStatus`	The CCID compliant slot status register is returned by this parameter. See the definition of `TISO7816StatusReg` for meaning of the different bit fields.
`byte* bError`	The CCID compliant error code is returned by this parameter.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.1.3 Card Deactivation

This function shall be used to deactivate and power off the card. When this function was called on TWN4 SmartCard reader, a subsequent call of `IccPowerOn()` leads to a cold reset of the card.

```
bool ISO7816_IccPowerOff(int Channel, TISO7816SlotStatus* SlotStatus);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`TISO7816SlotStatus* SlotStatus`	The card slot status is returned by this parameter. See the definition of `TISO7816SlotStatus` for meaning of each member.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.1.4 Set Communication Settings

This function shall be used to assign new communication settings to the respective card slot. After calling this function, the communication parameters which have been negotiated with the card during Protocol And Parameter Selection (PPS) become valid. For issuing a PPS, please refer to the function `ISO7816_Transceive`. Specific communication parameters must be obtained from the ATR, for detailed information refer to standard ISO7816-3.

```
bool ISO7816_SetCommSettings
    (
    int Channel,
```

```
const TISO7816CommSettings* CommSettings
);
```

Parameters:

`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`const TISO7816CommSettings* CommSettings`	The new communication settings are passed by this parameter. See the definition of `TISO7816CommSettings` for meaning of each member.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
`byte Protocol`	8	Specify the protocol to be used. Possible values are: `ISO7816_PROTOCOL_T0` and `ISO7816_PROTOCOL_T0`.
`byte Freq`	8	Specify the clock frequency which shall be applied to the card. Chose one of the predefined constants `ISO7816_FREQUENCY_1000000`, `ISO7816_FREQUENCY_1250000`, `ISO7816_FREQUENCY_1875000`, `ISO7816_FREQUENCY_2500000`, `ISO7816_FREQUENCY_3750000`, `ISO7816_FREQUENCY_5000000`, `ISO7816_FREQUENCY_7500000` or `ISO7816_FREQUENCY_15000000`.
`uint16_t F`	16	Specify a non-ISO value for F.
`uint16_t D`	16	Specify a non-ISO value for D.
`union TProtocolData ProtocolData`	56	See definition of `TProtocolData` for details.

Table 29.3: Definition of `TISO7816CommSettings`

Members	Length (Bits)	Description
`TProtocolDataT0 T0`	40	See definition of `TProtocolDataT0` for details.
`TProtocolDataT1 T1`	56	See definition of `TProtocolDataT1` for details.

Table 29.4: Definition of `TProtocolData`

Members	Length (Bits)	Description
byte bmFindexDindex	8	Bit 7-4: FI, Index into table 7 of ISO/IEC 7816-3:2006 selecting a clock rate conversion factor. Bit 3-0: DI, Index into table 8 of ISO/IEC 7816-3:2006 selecting a baud rate conversion factor. This value shall be obtained from TA1 of the ATR.
byte bmTCCKST0	8	This value shall be set to 00h.
byte bGuardTimeT0	8	Extra Guardtime between two characters. Add 0 to 254 etu to the normal guardtime of 12 etu. FFh is the same as 00h. This value shall be obtained from TC1 of the ATR.
byte bWaitingIntegerT0	8	Waiting time between transmission of a command and reception of the response. This value is specified in TC2 of the ATR. If TC2 is not present, the default value is 10.
byte bClockStop	8	This value shall be set to 00h.

Table 29.5: Definition of TProtocolDataT0

Members	Length (Bits)	Description
byte bmFindexDindex	8	Bit 7-4: FI, Index into table 7 of ISO/IEC 7816-3:2006 selecting a clock rate conversion factor. Bit 3-0: DI, Index into table 8 of ISO/IEC 7816-3:2006 selecting a baud rate conversion factor. This value shall be obtained from TA1 of the ATR.
byte bmTCCKST1	8	This value shall be set to 00h.
byte bGuardTimeT1	8	Extra Guardtime (0 to 254 etu between two characters). If value is FFh, then guardtime is reduced by 1 etu. This value shall be obtained from TC1 of the ATR.
byte bWaitingIntegerT1	8	Bit 7-4: BWI, values 0-9 valid. Bit 3-0: CWI, values 0-Fh valid. This value is specified in the first TB for T=1 in the ATR.
byte bClockStop	8	This value shall be set to 00h.
byte bIFSC	8	Size of negotiated IFSC in bytes. This value is specified in the first TA for T=1 in the ATR.
byte bNadValue	8	This value shall be set to 00h.

Table 29.6: Definition of TProtocolDataT1

29.1.5 Transparent Data Transmission

This function shall be used for byte-wise communication with the card.

```
bool ISO7816_Transceive
    (
    int Channel,
    const byte* TX,
    int LenTX,
    byte* RX,
    int* LenRX,
    int MaxRXByteCnt,
    TISO7816StatusReg* bStatus,
    byte* bError
    );
```

Parameters:

`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`const byte* TX`	This buffer holds the data which shall be transmitted to the card.
`int LenTX`	This parameter specifies the data-length in bytes which shall be transmitted to the card.
`byte* RX`	This buffer holds the data which was read from the card. Take care for adequate dimensioning.
`int* LenRX`	After successful completion of this function, this parameter holds the number of bytes read from the card.
`int MaxRXByteCnt`	This parameter holds the array-size of `RX` in bytes.
`TISO7816StatusReg* bStatus`	The CCID compliant slot status register is returned by this parameter. See the definition of `TISO7816StatusReg` for meaning of the different bit fields.
`byte* bError`	The CCID compliant error code is returned by this parameter.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.1.6 Exchange Of APDUs

This function shall be used for APDU exchange based on T=0/T=1 protocol according to ISO7816-3.

```
bool ISO7816_ExchangeAPDU
    (
    int Channel,
    const TISO7816_ProtocolHeader* Header,
    const byte* TXData,
    int TXByteCnt,
    byte* RXData,
    int* RXByteCnt,
    int MaxRXByteCnt,
    uint16_t* StatusWord
    );
```

Parameters:

`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`const TISO7816_ProtocolHeader *Header`	This parameter holds basic APDU information.
`const byte* TXData`	This buffer holds the data field of the APDU.
`int TXByteCnt`	This parameter specifies the data-length in bytes of the data-field.
`byte* RXData`	This buffer holds the data-field of the received APDU.
`int* RXByteCnt`	After successful completion of this function, this parameters holds the data-field size of the received APDU.
`int MaxRXByteCnt`	This parameter holds the array-size of `RXData` in bytes.
`uint16_t* StatusWord`	This parameter holds the status word received from the card.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
`byte CLA`	8	This member holds the CLA-value.
`byte INS`	8	This member holds the INS-code.
`byte P1`	8	This member holds the parameter P1.
`byte P2`	8	This member holds the parameter P2.
`uint16_t Lc`	16	This member holds Lc which defines the size of the following data-field.
`uint16_t Le`	16	This member holds Le which defines the maximum expected size of the response data-field.
`struct TISO7816_ProtocolHeaderFlags Flags`	8	This member holds additional APDU information.

Table 29.7: Definition of `TISO7816_ProtocolHeader`

Members	Length (Bits)	Description
`byte LePresent`	1	If set to `true`, `Le` is transmitted.
`byte ExtendedAPDU`	1	If set to `true`, this APDU is sent as Extended APDU.
`byte RFU`	6	Reserved for future use.

Table 29.8: Definition of `TISO7816_ProtocolHeaderFlags`

29.1.7 Examples

29.1.7.1 PPS Example

The following example shows how to make a PPS with an ISO7816 card.

```
bool ISO7816_PPS(int Channel, byte Protocol, byte* bmFindexDindex)
{
    byte Cmd[4];
    byte Res[4];
    int TxByteCnt;
    int RxByteCnt;
    TISO7816StatusReg bStatus;
    byte bError;


    // PPS always starts with 0xFF
    Cmd[0] = 0xFF;
    // The second byte stores the desired protocol
    Cmd[1] = Protocol & 0x0F;
    // Is bmFindexDindex present?
    if (bmFindexDindex != NULL)
    {
        // Yes, prepare the command accordingly
        Cmd[1] |= 0x10;
        Cmd[2] = *bmFindexDindex;
        // Calculate the BCC over all command bytes
        Cmd[3] = Cmd[0] ^ Cmd[1] ^ Cmd[2];
        TxByteCnt = 4;
    }
    else
    {
        // FindexDindex is not present, calculate only BCC
        Cmd[2] = Cmd[0] ^ Cmd[1];
        TxByteCnt = 3;
    }
    // Send PPS request to the card, get response
    if (!ISO7816_Transceive(Channel, Cmd, TxByteCnt, Res,
                            &RxByteCnt, sizeof(Res), &bStatus, &bError))
        return false;
    // We expect the card to echo the request in its response
    if (RxByteCnt != TxByteCnt)
        return false;
    return memcmp(Cmd, Res, RxByteCnt) == 0;
}
```

29.1.7.2 Communication Example

The following example shows how to prepare a ISO7816 card for communication at T=1 protocol and exchange APDUs.

```
byte ATR[32];
int ATRByteCnt;

TISO7816SlotStatus SlotStatus;
TProtocolDataT1 ProtocolDataT1;
```

```
TISO7816_ProtocolHeader Header;
byte TXData[128];
byte RXData[128];
int RXByteCnt;
uint16_t SW12;

// We want to use T=1 protocol with the following non-default values
ProtocolDataT1.bmFindexDindex = 0x98;
ProtocolDataT1.bmTCCKST1 = 0;
ProtocolDataT1.bGuardTimeT1 = 0xFF;
ProtocolDataT1.bmWaitingIntegersT1 = 0x55;
ProtocolDataT1.bClockStop = 0;
ProtocolDataT1.bIFSC = 0xFE;
ProtocolDataT1.bNadValue = 0x00;

MainLoop:
while (true)
{
    // Is a card inserted in CHANNEL_SC1?
    if (!ISO7816_GetSlotStatus(CHANNEL_SC1, &SlotStatus))
        goto MainLoop;
    // Card slot empty?
    if (SlotStatus.bStatus.bmICCStatus == ISO7816_NOICCPRESENT)
        goto MainLoop;
    // Perform activation of the card and receive ATR
    if (!ISO7816_IccPowerOn
        (
        CHANNEL_SC1,
        ATR,
        &ATRByteCnt,
        sizeof(ATR),
        ISO7816_POWERSELECT_5V,
        &SlotStatus.bStatus,
        &SlotStatus.bError
        ))
        goto MainLoop;
    // We expect the card to be present and active
    if (SlotStatus.bStatus.bmICCStatus != ISO7816_ICCPRESENTANDACTIVE)
        goto MainLoop;
    // Perform PPS for T=1 protocol
    if (!ISO7816_PPS(CHANNEL_SC1, ISO7816_PROTOCOL_T1,
                     &ProtocolDataT1.bmFindexDindex))
        goto MainLoop;

    // Let's prepare our APDU. We want to select the Masterfile (MF)
    // of a PKI card by its SFI (0x3F00).
    Header.CLA = 0x00;
    Header.INS = 0xA4;
    Header.P1  = 0x00;
    Header.P2  = 0x00;
    Header.Lc  = 0x0002;
    Header.Le  = 0x0000;
    Header.Flags.LePresent = true;
    Header.Flags.ExtendedAPDU = false;
    TXData[0]  = 0x3F;
    TXData[1]  = 0x00;
```

```
    // Exchange the APDU
    if (!ISO7816_ExchangeAPDU(CHANNEL_SC1, &Header, TXData, Header.Lc,
                              RXData, &RXByteCnt, sizeof(RXData), &SW12))
        goto MainLoop;
    // Check status word of the received APDU
    if (SW12 == 0x9000)
    {
        // Further APDUs may follow...
        DoSomething();
    }
}
```

29.2 SLE Memory Cards

This chapter shows how to use TWN4 with contact based memory cards such as SLE44xx or compatible cards. In order to query the card slot insertion state, the function `ISO7816_GetSlotStatus` can be used.

29.2.1 Get ATR

Use this function to retrieve the ATR (Answer To Reset) from an inserted card.

```
bool SLE_GetATR(int Channel, byte* ATR);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`byte* ATR`	The card's ATR is returned by this buffer. The function always returns 4 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.2 Read Main Memory

Use this function to read data from the main memory.

```
bool SLE_ReadMainMemory(int Channel, int Address, byte* Data, int ByteCnt);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Address`	Specify the start address in memory for reading.
`byte* Data`	This buffer holds the data read from the card. Take care for proper dimensioning.
`int ByteCnt`	Specify the number of bytes to be read.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.3 Write Main Memory

Use this function to write one byte of data to the main memory.

```
bool SLE_UpdateMainMemory(int Channel, int Address, byte Value);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Address`	Specify the address in memory to be written.
`byte Value`	Specify the data byte to be written.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.4 Read Security Memory

Use this function to read out the four bytes of Security Memory.

```
bool SLE_ReadSecurityMemory(int Channel, byte* SecMemData);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`byte* SecMemData`	This buffer holds the Security Memory data read from the card. The function always returns 4 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.5 Write Security Memory

Use this function to write one byte of data to the Security Memory.

```
bool SLE_UpdateSecurityMemory(int Channel, int Address, byte SecMemData);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Address`	Specify the address in Security Memory to be written.
`byte SecMemData`	Specify the data byte to be written.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.6 Read Protection Memory

Use this function to read out the four bytes of Protection Memory.

```
bool SLE_ReadProtectionMemory(int Channel, byte* ProtMemData);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`byte* ProtMemData`	This buffer holds the Protection Memory data read from the card. The function always returns 4 bytes.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.7 Write Protection Memory

Use this function to write one byte of data to the Protection Memory.

```
bool SLE_WriteProtectionMemory(int Channel, int Address, byte ProtMemData);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Address`	Specify the address in Protection Memory to be written.
`byte ProtMemData`	Specify the data byte to be written.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.2.8 Compare Verification Data

Use this function to transmit one byte of verification input to the card.

```
bool SLE_CompareVerificationData(int Channel, int Address, byte VerificationData);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Address`	Specify the address of verification data byte.
`byte VerificationData`	Specify the verification data byte to be transfered to the card.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.3 I2C Memory Cards

This chapter shows how to use TWN4 with contact based I2C memory cards. In order to query the card slot insertion state, the function `ISO7816_GetSlotStatus` can be used.

29.3.1 Read Memory

Use this function to read data from the memory.

```
bool I2CCard_Read(int Channel, int Addr, byte* Data, int ByteCnt);
```

Parameters:	
`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Addr`	Specify the start address in memory for reading.
`byte* Data`	This buffer holds the data read from the card. Take care for proper dimensioning.
`int ByteCnt`	Specify the number of bytes to be read.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

29.3.2 Write Memory

Use this function to write data to the memory.

```
bool I2CCard_Write(int Channel, int Addr, const byte* Data, int ByteCnt);
```

Parameters:

`int Channel`	Specify a communication channel by this parameter. Valid values are `CHANNEL_SAM1` through `CHANNEL_SAM4` or `CHANNEL_SC1`, use one of these predefined constants.
`int Addr`	Specify the start address in memory for the write operation.
`const byte* Data`	Specify data to be written.
`int ByteCnt`	Specify the number of bytes to be written.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

30 Cryptographic Operations

The cryptographic API incorporates methods for encryption/decryption of data, these are Triple-DES (Data Encryption Standard) or AES (Advanced Encryption Standard). TDES is available in two versions that support different key-lengths: 128 bit (TDES2K) and 192 bit (TDES3K).
The implementation of TDES is based on FIPS PUB 46-3. The method always operates on entire data blocks of 8 bytes. The DES algorithm is passed three times for one TDES operation. In case of TDES2K, the 128 bit key is hereby split into two parts: K1 and K2. In case of TDES3K, the 192 bit key is split into three parts: K1, K2 and K3.
The implementation of AES is based on FIPS PUB 197. The method always operates on entire data blocks of 16 bytes, the key-length is 128 bit.

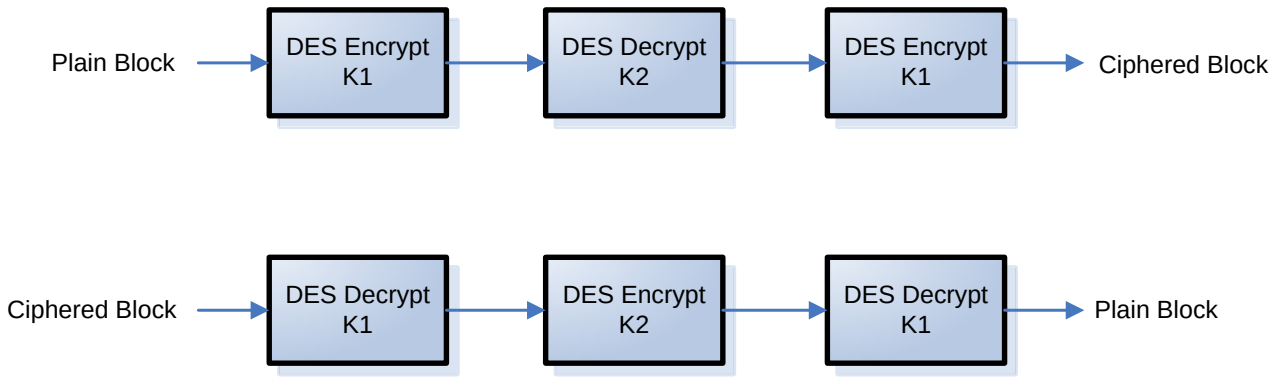


Figure 30.1: TDES2K Operation

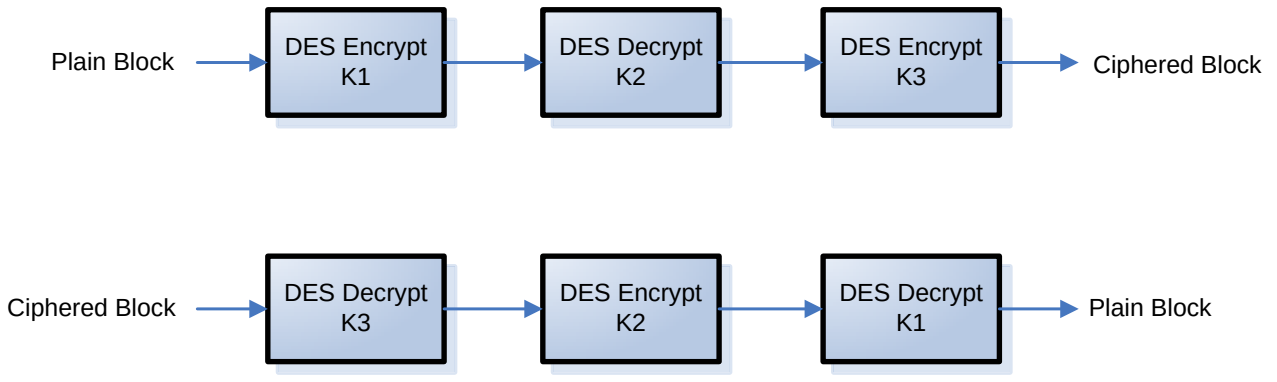


Figure 30.2: TDES3K Operation

The cryptographic API may be used to simply encrypt/decrypt a single block or to encrypt/decrypt a chain of blocks using the CBC-method (Ciphered Block Chaining).
In CBC mode, every ciphering operation depends on the foregoing step, this is achieved by involving the

so-called Init Vector IV. The first CBC-operation usually works with an Init Vector that is set to zero.
For encryption, a plain data block P is logically XOR-ed with this Init Vector before it comes to encryption. The result is a ciphered block C which serves as Init Vector for the next operation. See the schematic below for details:

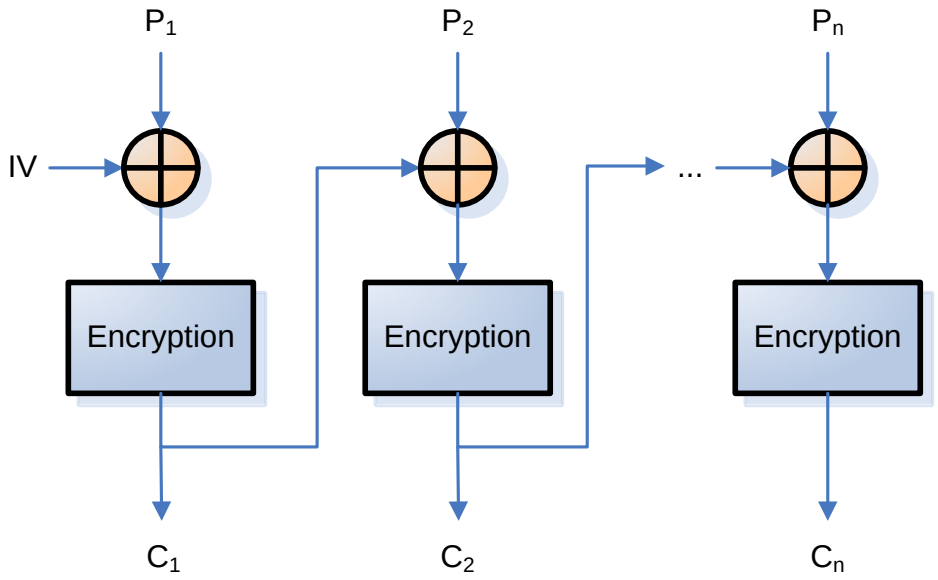


Figure 30.3: CBC Enciphering scheme

If a ciphered block C is decrypted, the result is logically XOR-ed with the Init Vector. See the schematic below for details:

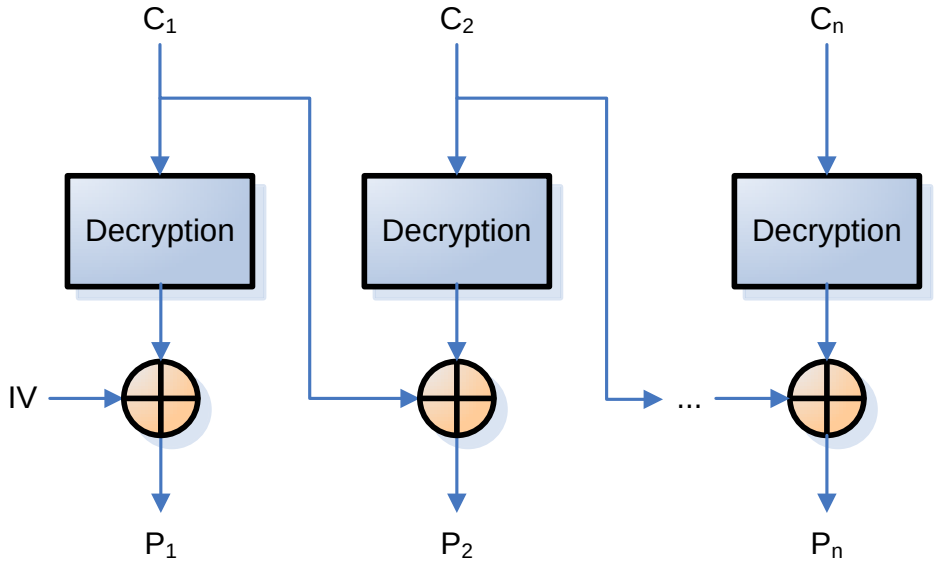


Figure 30.4: CBC Deciphering scheme

30.1 Initialization

The cryptographic API has to be initialized before it can be used. During initialization the key is passed to the cryptographic method and assigned to a cryptographic environment. After initialization the functions for encryption and decryption are set up for the desired cryptographic mode. If a cryptographic environment is configured for CBC-operation, the internally managed Init Vector is automatically reset to zero.

```
void Crypto_Init
    (
    int CryptoEnv,
    int CryptoMode,
    const byte* Key,
    int KeyByteCnt
    );
```

Parameters:

`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`int CryptoMode`	Specify the mode of cryptographic operation. Choose either one of the predefined non-CBC constants `CRYPTOMODE_3DES`, `CRYPTOMODE_3K3DES`, `CRYPTOMODE_AES128` or one of the pre-defined CBC constants `CRYPTOMODE_CBC_DES`, `CRYPTOMODE_CBC_DFN_DES`, `CRYPTOMODE_CBC_3DES`, `CRYPTOMODE_CBC_DFN_3DES`, `CRYPTOMODE_CBC_3K3DES`, `CRYPTOMODE_CBC_AES128`.
`const byte* Key`	The key is passed by this parameter. Depending on the specified crypto mode, the key-length is either 16 or 24 bytes.
`int KeyByteCnt`	Specify the length of the key in bytes.
Return:	This function has no return value.

30.2 Encrypt

Use this function to encrypt a plain block of data.

```
void Encrypt
    (
    int CryptoEnv,
    const byte* PlainBlock,
    byte* CipheredBlock,
    int BlockByteCnt
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`const byte* PlainBlock`	Pointer to the array, that contains the plain data block to be encrypted.
`byte* CipheredBlock`	Pointer to the array, that receives the encrypted data block. Take care for proper dimensioning.
`int BlockByteCnt`	Specify the number of bytes of a block.
Return:	This function has no return value.

30.3 Decrypt

Use this function to decrypt an encrypted block of data.

```
void Decrypt
    (
    int CryptoEnv,
    const byte* CipheredBlock,
    byte* PlainBlock,
    int BlockByteCnt
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
`const byte* CipheredBlock`	Pointer to the array, that holds the encrypted data block.
`const byte* PlainBlock`	Pointer to the array, that receives the decrypted data block. Take care for proper dimensioning.
`int BlockByteCnt`	Specify the number of bytes of a block.
Return:	This function has no return value.

30.4 Reset Init Vector

Use this function to manually reset the internally managed Init Vector of a cryptographic environment to zero.

```
void CBC_ResetInitVector
    (
    int CryptoEnv
    );
```

Parameters:	
`int CryptoEnv`	Specify a cryptographic environment by this parameter. The valid range is `CRYPTO_ENV0` to `CRYPTO_ENV3`, use one of these predefined constants.
Return:	This function has no return value.

31 Storage Functions

This chapter describes functions for accessing the storage of TWN4.

The storage memory is part of the internal flash of the main control unit (MCU) of TWN4. The gross amount of this storage is 48kByte. Due to segmentation of the memory and further control mechanisms, after deduction the memory size is 18kByte.

Before first use of the storage, the storage must be formatted. The appropriate system function for doing so is `FSFormat`.

In order to gain access to the storage memory, the file system must be initialized and connected to the internal flash. This can be achieved with the system function `FSMount`.

Why is a separate mount needed to gain access to the storage memory?

The reason for a separate mount is, that there could be a reasonable amount of time required in order to start the file system. Background is, that depending on the state of the file system, additional activities must be started, before access of the storage memory is possible. There is especially the situation, which can occur, if last file operation were interrupted by a unplanned power fail. This can lead to the situation, that the file system must be reset to the state, before the interrupted file operation was started. This clean-up is done by function `FSMount`

The structure of the storage memory is similar to a none-hierarchical file system. Following points must be known:

- Data is structured in files.
- Files are indicated by a file ID. The file ID is any 32 bit number.
- It is possible to iterate through the existing files and thus list the files stored in the memory.
- There is a maximum number of files, which can be stored in the memory. This maximum number is 16.
- In order to read from or write to files, appropriate system functions are available. In order to start a file operation, the file must be opened for appropriate file operation. The maximum number of files, which can be kept opened at a time is 4.
- File operations are kept atomic. This means: If a change to a file (some kind write operation) is interrupted by a power fail, the file system returns to the state, where the change began.

31.1 Management Functions

31.1.1 FSMount

Before any access to files can be performed, the appropriate file system must be mounted. Following steps are performed by function `FSMount`:

- Check, if the specified volume contains a valid file system.

- Check, if there is a not completed file operation.
- If applicable, unwind file system to the point where not completed file operation was started.
- Finally, create a logical link between volume and file system.

```
bool FSMount(int StorageID,int Mode)
```

Parameters:	
`int StorageID`	Specifies the volume, which should be mounted. Currently, there is one storage available, the internal flash. The appropriate definition for this storage is `SID_INTERNALFLASH`
`int Mode`	Specifies the mode in which the volume is mounted. This can be `FS_MOUNT_NONE` (equivalent to a unmount), `FS_MOUNT_READONLY` (no write access to storage possible)or `FS_MOUNT_READWRITE` (full read/write access).
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

31.1.2 FSFormat

This function prepares the storage memory of TWN4 for further file operations.

— WARNING — WARNING — WARNING —

All data, which is stored on the file system will be irrecoverable deleted by calling this function!

```
bool FSFormat(int StorageID,int MagicValue)
```

Parameters:	
`int StorageID`	Specifies the volume, which should be formatted.
`int MagicValue`	In order to avoid accidantely format of a volume, an appropriate parameter for `MagicValue` must be specified. There is a definition for this magic value, which is `FS_FORMATMAGICVALUE`.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

31.2 File Functions

31.2.1 FSOpen

This function must be called in order to begin any read or write operation from/to a file.

Following definitions for the parameter mode are valid:

FS_READ	Open a file for read access. If the file not exists, an error is generated. The position of the read pointer is set to zero, thus to the start of the file.
FS_WRITE	Open a file for write access. An empty file is created independently of if the file already exists or not, thus content of an earlier version of that file will be deleted.
FS_APPEND	Open a file for write access. If the file does not exist, a new file is created. If the file already exists, the file pointer is moved to the end of the file, which means, that newly written data is appended to data of existing file.

Following further considerations:

- A file can be opened one time in mode FS_WRITE or FS_APPEND, but never, if it is already opened by any other file operation.
- A file can be opened many times in mode FS_READ, but never, if it is already opened in mode FS_WRITE or FS_APPEND by another file operation.

```
bool FSOpen(int FileEnv,int StorageID,uint32_t FileID,int Mode)
```

Parameters:	
int FileEnv	Specifies the environment to be used for the file operation. Up to four file operations can be opened at a time. The appropriate defintions for these environments are FILE_ENV0 - FILE_ENV3.
int StorageID	Specifies the storage on which the file resides. Currently, this parameter can be SID_INTERNALFLASH only.
uint32_t FileID	Specifies the ID of a file. The file ID is a reduced version of file name and be understood as such. File ID is an integer number from 1 to $2^{32}-1$, thus 0x00000000 to 0xFFFFFFFF.
int Mode	Specifies, how the file is accessed (see above).
Return:	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

X

31.2.2 FSClose

This function is used to terminate a file operation. Several actions are taken, when this function is called:

- Pending data is written to the storage system.
- If this is the last file being closed, the file system is finalized in terms, that the even loss of the power will restore this now achieved state.

```
bool FSClose(int FileEnv)
```

Parameters:	
int FileEnv	Specifies the environment to be used for the file operation.
Return:	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

31.2.3 FSCloseAll

This function is closing all opened file operations throughout all mounted storages. This function avoids keeping track of opened file operations.

```
void FSCloseAll(void)
```

Parameters:	None.
Return:	None.

31.2.4 FSSeek

Read and write operations from/to a file are implemented via a file pointer, which references the point, from which next data is read or where next data is written. With this function, the file pointer can be moved throughout a file and furthermore in relation to a specific point of the file.

FS_POSABS	Move file position in relation to the start of the file. This results in a move of the file pointer to an absolute position.
FS_POSREL	Move the file pointer in relation to the current position. This allows an easy skip of a number of bytes of the file.
FS_POSEND	Move the file pointer in relation to the end of the file. This allows to move to the end of the file without knowledge and independent of the length of a file.

```
bool FSSeek(int FileEnv,int Origin,int Pos)
```

Parameters:	
int FileEnv	Specifies the environment to be used for the file operation.
int Origin	Specifies the reference point, from which the new file position is calculated (see above).
int Pos	Specifies the number of bytes in relation to the reference point. A negative value is treated as position before reference point, a positive value is treated as position behind the reference point.
Return:	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

31.2.5 FSTell

This function returns the position of the file pointer in relation to a reference point. Please note that in consequence, specifying FS_POSREL as origin must always return the value zero.

```
bool FSTell(int FileEnv,int Origin,int *Pos)
```

Parameters:	
int FileEnv	Specifies the environment to be used for the file operation.
int Origin	Specifies the reference point, under which the current position is calculated (see function FSSeek).
int *Pos	A pointer to an integer, which will receive the value of the position.
Return:	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

31.2.6 FSReadBytes

Read bytes from a file, which has been opened in mode FS_READ before. Use function FSOpen to open the file accordingly.

The function generates the error ERR_ENDOFFILE, if less than the requested number of bytes were read from the file or if there are no more bytes left to be read from the file.

```
bool FSReadBytes(int FileEnv,void *Data,int ByteCount,int *BytesRead)
```

Parameters:	
int FileEnv	Specifies the environment to be used for the file operation.
void *Data	Pointer to an array of bytes, which receives read data.
int ByteCount	Number of bytes, which should be read from the file.
int *BytesRead	Pointer to an integer, which receives the number of actually read bytes. The received value is valid even if the function returns with an error.
Return:	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

31.2.7 FSWriteBytes

Write bytes to a file, which has been opened in mode FS_WRITE or FS_APPEND before. Use function FSOpen to open the file accordingly.

```
bool FSWriteBytes(int FileEnv,const void *Data,int ByteCount,int *BytesWritten)
```

Parameters:	
int FileEnv	Specifies the environment to be used for the file operation.
const void *Data	Pointer to an array of bytes, which contains data to be written.
int ByteCount	Number of bytes, which should be written to the file.
int *BytesWritten	Pointer to an integer, which receives the number of actually written bytes. The received value is valid even if the function returns with an error.
Return:	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

31.3 Directory Functions

31.3.1 FSFindFirst

The functions FSFindFirst/FSFindNext implement the possibility to enumerate the files contained in a files system. In order to begin enumeration of files the function FSFindFirst must be called.

The members of a directory entry are stored in a structure of type TFileInfot. The members of the structure are:

ID	The file ID.
Length	The length of the file.

```
bool FSFindFirst(int StorageID,TFileInfo *pFileInfo)
```

Parameters:	
int StorageID	Storage ID of the file system, where files should be enumerated.
TFileInfo *pFileInfo	Pointer to a structure of type TFileInfo which receives a directory entry.
Return:	If the operation was successful, the return value is true, otherwise it is false. If no directory entry was found the error code ERR_FILENOTFOUND is generated. The concrete error code can be retrieved with system function GetLastError.

31.3.2 FSFindNext

The functions FSFindFirst/FSFindNext implement the possibility to enumerate the files contained in a files system. In order to continue enumeration, once first entry has been retrieved with function FSFindFirst, the function FSFindNext must be called.

```
bool FSFindNext(TFileInfo *pFileInfo)
```

Parameters:	
TFileInfo *pFileInfo	Pointer to a structure of type TFileInfo which receives a directory entry.
Return:	If the operation was successful, the return value is true, otherwise it is false. If no directory entry was found the error code ERR_FILENOTFOUND is generated. The concrete error code can be retrieved with system function GetLastError.

31.3.3 FSDelete

Use function FSDelete to delete files from the file system. A file, which is currently opened can not be deleted.

```
bool FSDelete(int StorageID,uint32_t FileID)
```

Parameters:	
`int StorageID`	Storage ID of the file in question.
`uint32_t FileID`	File ID of the file to be deleted.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

31.3.4 FSRename

Use function `FSRename` to rename files on the file system.

```
bool FSRename(int StorageID,uint32_t OldFileID,uint32_t NewFileID)
```

Parameters:	
`int StorageID`	Storage ID of the file in question.
`uint32_t OldFileID`	Current file ID of the file to be renamed.
`uint32_t NewFileID`	Future file ID of the file to be renamed.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

31.4 Miscellaneous Functions

31.4.1 FSGetStorageInfo

Function `FSGetStorageInfo` allows to retrieve information regarding a storage.

```
bool FSGetStorageInfo(int StorageID,TStorageInfo *pStorageInfo)
```

Parameters:	
`int StorageID`	ID of the storage in question.
`TStorageInfo *pStorageInfo`	Pointer to a structure of type `TStorageInfo`, which receives the requested information.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

The structure `TStorageInfo` is defined as follows:

```
typedef struct
{
    byte ID;
    uint32_t Size;
    uint32_t Free;
} TStorageInfo;
```

where:

byte ID	ID of the storage in question.
uint32_t Size	Size in bytes of the storage.
uint32_t Free	Number of free bytes in the storage.

31.5 Examples

This is an example for a function, which reads a complete file from the file system. The file system must have been mounted before with function FSMount.

```
bool ReadFile1(uint32_t FileID,byte *Data,int *FileLength,int MaxFileLength)
{
    if (!FSOpen(FILE_ENV0,SID_INTERNALFLASH,FileID,FS_READ))
        return false;
    FSReadBytes(FILE_ENV0,Data,MaxFileLength,FileLength);
    int LastError = GetLastError();
    FSClose(FILE_ENV0);
    if (LastError != ERR_NONE && LastError != ERR_ENDOFFILE)
        return false;
    // Function was successfully completed
    return true;
}
```

Here is an example for a function, which reads a complete file from the file system but in portions of 256 bytes. This might be useful, if the implementation is actually done on a host, which is doing system calls indirectly via TWN4 Simple Protocol. The file system must have been mounted before with function FSMount.

```
bool ReadFile2(uint32_t FileID,byte *Data,int *Length,int ExpectedLength)
{
    *Length = 0;
    if (!FSOpen(FILE_ENV0,SID_INTERNALFLASH,FileID,FS_READ))
        return false;
    bool ReadSuccess;
    int RemainingBytes = ExpectedLength;
    do
    {
        if (RemainingBytes == 0)
        {
            FSClose(FILE_ENV0);
            return true;
        }
        const int BlockSize = 256;
        int BytesToRead = RemainingBytes;
        if (BytesToRead > BlockSize)
            BytesToRead = BlockSize;
        int BytesRead;
        ReadSuccess = FSReadBytes(FILE_ENV0,Data,BytesToRead,&BytesRead);
        Data += BytesRead;
        *Length += BytesRead;
        RemainingBytes -= BytesRead;
    }
    while (ReadSuccess);
    int LastError = GetLastError();
    FSClose(FILE_ENV0);
    if (LastError != ERR_NONE && LastError != ERR_ENDOFFILE)
```

```
        return false;
    // Function was successfully completed
    return true;
}
```

Here is an example for a function, which writes a complete file to the file system in portions of 256 bytes. This might be useful, if the implementation is actually done on a host, which is doing system calls indirectly via TWN4 Simple Protocol. The file system must have been mounted before with function `FSMount`.

```
bool WriteFile(uint32_t FileID,byte *Data,int Length)
{
    if (!FSOpen(FILE_ENV0,SID_INTERNALFLASH,FileID,FS_WRITE))
        return false;
    bool WriteSuccess;
    int RemainingBytes = Length;
    do
    {
        if (RemainingBytes == 0)
        {
            FSClose(FILE_ENV0);
            return true;
        }
        const int BlockSize = 256;
        int BytesToWrite = RemainingBytes;
        if (BytesToWrite > BlockSize)
            BytesToWrite = BlockSize;
        int BytesWritten;
        WriteSuccess = FSWriteBytes(FILE_ENV0,Data,BytesToWrite,&BytesWritten);
        Data += BytesWritten;
        RemainingBytes -= BytesWritten;
    }
    while (WriteSuccess);
    int LastError = GetLastError();
    FSClose(FILE_ENV0);
    if (LastError != ERR_NONE)
        return false;
    // Function was successfully completed
    return true;
}
```

32 System Parameters

The TWN4 App-system provides methods of setting up paramaters before or during runtime of Apps.

- In order to set up parameters before the App is started, a so-called Manifest can be specified as part of an App.
- In order to set up parameters during normal execution of an App there is the system function `SetParameters`.

This section describes the specification of a Manifest and all available parameters. See chapter "System Functions" for a description of function `SetParameters`.

32.1 TLV Format

Parameters for a Manifest or the system function `SetParameters` are specified in the TLV format. The TLV format specifies a chain of parameters with variable type and length. This format must follow following rules:

- Every entry (except the last entry) is a sequence of 3 items. The 3 items are 'Type', 'Length' and 'Value'.
- The name of the parameter is associated to 'Type', the length of
- the value is associated to 'Length' and the value itself is associated to 'Value'
- The TLV list must be terminated with an item consisting of just the type. This type must contain the value `TLV_END`.

32.2 Manifest

The intention for specifying a Manifest as part of an App could be to avoid opening of communication channels in order to further reduce current consumption. Another could be to modify behaviour of the USB section of TWN4.

The specification of a Manifest is pretty simple:

Define an array of bytes with the key-name Manifest. This will point the firmware of TWN4 to the position where the parameters of interest are stored. Here is an example:

Example:

```
// This sample demonstrates the specification of a Manifest:
const byte *Manifest =
{
    OPEN_PORTS, 1, OPEN_PORT_USB_MSK,   // Open USB channel only
    TLV_END                             // End of TLV
};
```

No further action is required.

32.3 Available Parameters

Here is a list of all parameters, which are supported:

Type (Parameter)	Length	Value
TLV_END	N/A	N/A
OPEN_PORTS	1	Bitwise OR of one or more of the following definitions: OPEN_PORT_USB_MSK OPEN_PORT_COM1_MSK OPEN_PORT_COM2_MSK
EXECUTE_APP	1	EXECUTE_APP_AUTO EXECUTE_APP_ALWAYS
INDITAG_READMODE	1	INDITAG_READMODE_1 INDITAG_READMODE_2
COTAG_READMODE	1	COTAG_READMODE_HASH COTAG_READMODE_1 COTAG_READMODE_2
COTAG_VERIFY	1	COTAG_VERIFY_OFF COTAG_VERIFY_ON
HONEYTAG_READMODE	1	HONEYTAG_READMODE_HASH HONEYTAG_READMODE_1
ICLASS_READMODE	1	ICLASS_READMODE_UID ICLASS_READMODE_PAC
AT55_BITRATE	1	8 to 128 as multiple of 2
AT55_OPTIONS	1	One of the following definitions: AT55_OPT_SEQUENCENONE AT55_OPT_SEQUENCETERMINATOR AT55_OPT_SEQUENCESTARTMARKER
CCID_MAXSLOTINDEX	1	Specify index of last logical CCID slot
HITAG1S_T0	1	Values from 14 to 40
HITAG1S_T1	1	Values from 14 to 40
HITAG1S_TGAP	1	Values from 2 to 14
HITAG2_T0	1	Values from 14 to 40
HITAG2_T1	1	Values from 14 to 40
HITAG2_TGAP	1	Values from 2 to 14
ISO14443_BITRATE_TX	1	One of the following possible bitrates: ISO14443_BITRATE_106 ISO14443_BITRATE_212 ISO14443_BITRATE_424 ISO14443_BITRATE_848

Continued from last page:

ISO14443_BITRATE_RX	1	One of the following possible bitrates: ISO14443_BITRATE_106 ISO14443_BITRATE_212 ISO14443_BITRATE_424 ISO14443_BITRATE_848
USB_SUPPORTREMOTEWAKEUP	1	USB_SUPPORTREMOTEWAKEUP_OFF USB_SUPPORTREMOTEWAKEUP_ON
EM4102_OPTIONS	1	Bitwise OR of one or more of the following definitions: EM4102_OPTIONS_F64 EM4102_OPTIONS_F32
EM4150_OPTIONS	1	Bitwise OR of one or more of the following definitions: EM4150_OPTIONS_F64 EM4150_OPTIONS_F40
USB_SERIALNUMBER	1	USB_SERIALNUMBER_OFF USB_SERIALNUMBER_ON
USB_KEYBOARDREPEATRATE	1	Number of milliseconds per keyboard event
SEOS_TREATMENT	1	SEOS_TREATMENT_ICLASS SEOS_TREATMENT_ISO14443A
SUPPORT_CONFIGCARD	1	SUPPORT_CONFIGCARD_OFF SUPPORT_CONFIGCARD_ON
ISO14443_3_TDX_CRCCONTROL	1	0x00 or bitwise OR of one or more of the following definitions: ISO14443_3_TDX_CRCCTRL_TX ISO14443_3_TDX_CRCCTRL_RX
ISO7816_CONTROL	2	0x0000 or bitwise OR of one or more of the following definitions: ISO7816_HANDLE_ERROR_SIGNAL_ATR ISO7816_TRANSMIT_ERROR_CNT ISO7816_RECEIVE_ERROR_CNT ISO7816_VOLTAGE_SYNC_CARDS ISO7816_SUPPORT_EMVCO
PN5180_LPCD_THRESHOLD	1	Values from 0 to 255
PN5180_LPCD_SENSING_PERIOD	2	Values from 1 to 2690

33 System Errors

Here is a list of all error codes, which are generated by the firmware of TWN4. The error codes can be retrieved with function `GetLastError`.

In the current version of the firmware, storage functions (`FS...`) are generating such errors.

Error Code	Description
`ERR_NONE`	No error occured.
`ERR_OUTOFMEMORY`	The excution of a function required more memory than was available.
`ERR_ISALREADYINIT`	There was a try to initialize a system module, which already was initialized.
`ERR_NOTINIT`	There was a try to use a function from a module, which is not initialized.
`ERR_ISALREADYOPEN`	There was a try to open a system resource, which is already is open.
`ERR_NOTOPEN`	There was a try to use a system resource, which must be opened before usage.
`ERR_RANGE`	A specified parameters exceeded the valid range of values.
`ERR_PARAMETER`	A specified parameters is not in set of valid parameters.
`ERR_UNKNOWNSTORAGEID`	A storage ID was specified, which is not known by the firmware.
`ERR_WRONGINDEX`	A index was specified, which was out of the valid range.
`ERR_FLASHERASE`	The erase of a section of the flash failed.
`ERR_FLASHWRITE`	The write to the flash memory failed.
`ERR_SECTORNOTFOUND`	A sector of the file system was not found.
`ERR_STORAGEFULL`	All sectors of the file system are occupied.
`ERR_STORAGEINVALID`	There is an error in the file system.

ERR_TRANSACTIONLIMIT	The limit of changes in the file system is reached, which is possible within one transactions.
ERR_UNKNOWNFS	The file system on the specified storage is not supported by the current firmware.
ERR_FILENOTFOUND	The specified file was not found.
ERR_FILEALREADYEXISTS	The specified file already exists.
ERR_ENDOFFILE	The end of the file was reached. There is no more data to be read. Note: This error code is generated even the system function returned successful execution.
ERR_STORAGENOTFOUND	The specified storage was not found, e.g. because it is not mounted.
ERR_STORAGEALREADYMOUNTED	The specified storage is already mounted.
ERR_ACCESSDENIED	The access to a file was denied, e.g. write access to a file in a storage, which is mounted as read only.
ERR_FILECORRUPT	The specified file is corrupt in terms of a corrupted file system.
ERR_INVALIDFILEENV	The specified environment is invalid.
ERR_INVALIDFILEID	The specified file ID is invalid.
ERR_RESOURCELIMIT	The maximum number of available resources have bee occupied.

Please see file `twn4.sys.h` (which can be found in local directory `Tools\sys\` of the developer pack) for concrete numbers, which are behind the definitions.

34 Runtime Library

There is a couple of functions, which are not part of the firmware of TWN4. Instead, they are statically linked to the App.

There are several intentions for such functions:

- Provide functions instead of having similar code in each App.
- Provide an API at a higher level to simplify writing Apps.
- Maintain a degree of compatibility to TWN3.

34.1 Timer Functions

Include file: `apptools.h`

There are three functions, which implement a simple API, which allows triggering events after a specified time. The behaviour of the functions are similar to TWN3. Compared to TWN3, there is only one timer available. Therefore no timer ID must be specified. These timer functions are implemented using system function `GetSysTicks`.

34.1.1 StartTimer

Start the timer with a specified time.

```
void StartTimer(unsigned long Duration)
```

Parameters:	
`unsigned long Duration`	Time in milliseconds, till function `TestTimer` returns `true`.
Return:	None.

34.1.2 StopTimer

Stop the timer, thus function `TestTimer` will never return `true`.

```
void StopTimer(void);
```

Parameters:	None.
Return:	None.

34.1.3 TestTimer

Test, if the timer reached the timeout which was programmed by function `StartTimer`.

```
bool TestTimer(void);
```

Parameters:	None.
Return:	If the timeout has been reached, the function returns `true`, otherwise, it return `false`.

34.2 Host Communication

Include file: `apptools.h`

There are several function which implement a simplified interface for direct write to the host. The host is defined to be a communication channel, where all communication takes place. This removes the requirement to specify the communication channel every time when communication should take place.

For a more sophisticated kind of communication (binary, bidirectional), it is suggested to directly use the I/O functions from the firmware.

34.2.1 SetHostChannel

Specify the channel, where communication should take place. By default, the channel is determined by the connected communication cable, which is therefore either USB (TWN4 USB) or COM1 (TWN4 RS232).

```
void SetHostChannel(int Channel)
```

Parameters:	
`int Channel`	Specifies the communication channel to be used. This might be `CHANNEL_USB`, `CHANNEL_COM1`, `CHANNEL_COM2` or `CHANNEL_I2C` or `CHANNEL_NONE`. If `CHANNEL_NONE` is specified, channel will be choosen depending on connected communication cable.
Return:	None.

34.2.2 HostTestByte

Use this function to check if there is a byte available in the input buffer of the host-channel.

```
bool HostTestByte(void)
```

Parameters:	None.
Return:	If there is a byte available, the return value is `true`, otherwise it is `false`.

34.2.3 HostReadByte

Use this function to read a byte from the input buffer of the host-channel. If there is no byte available, the function blocks until there is one.

```
byte HostReadByte(void)
```

Parameters:	None.
Return:	The byte which was read from the input buffer.

34.2.4 HostTestChar

Test if a character is available from the host. The character can be read with function `HostReadChar`.

```
bool HostTestChar(void)
```

Parameters:	None.
Return:	`true` if at least one character arrived, otherwise `false`.

34.2.5 HostReadChar

Receive a character from the host. This is a blocking function. This means, it is waiting, till a character is available.

```
char HostReadChar(void)
```

Parameters:	None.
Return:	The character, which was read from the host.

34.2.6 HostWriteByte

Use this function to send one byte to the host through the actually configured host-channel. If the output buffer is completely occupied, the function blocks until there is enough space.

```
void HostWriteByte(byte Byte)
```

Parameters:	
`byte Byte`	The byte to be sent.
Return:	None.

34.2.7 HostWriteChar

Send a character to the host. This is a blocking function. This means, it is waiting, till there is storage in the output buffer, to transmit the character.

```
void HostWriteChar(char Char)
```

Parameters:	
`char Char`	The character to be sent to the host.
Return:	None.

34.2.8 HostWriteString

Send a string to the host. The string must be terminated with a null character. The string is sent without the null character.

```
void HostWriteString(const char *String)
```

Parameters:	
`const char *String`	Pointer to the string to be sent.
Return:	None.

34.2.9 HostWriteRadix

Send a number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteRadix(const byte *ID,int BitCnt,int DigitCnt,int Radix)
```

Parameters:	
`const byte *ID`	Pointer to the array of bytes.
`int BitCnt`	Number of bits stored in the array.
`int DigitCnt`	Number of output digits.
`int Radix`	Base for conversion from binary to ASCII. Use: • 2 for binary conversion • 8 for octal conversion • 10 for decimal conversion • 16 for hexadecimal conversion
Return:	None.

34.2.10 HostWriteBin

Send a binary number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteBin(const byte *ID,int BitCnt,int DigitCnt)
```

Parameters:	
`const byte *ID`	Pointer to the array of bytes.
`int BitCnt`	Number of bits stored in the array.
`DigitCnt`	Number of output digits.
Return:	None.

34.2.11 HostWriteDec

Send a decimal number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteDec(const byte *ID,int BitCnt,int DigitCnt)
```

Parameters:	
`const byte *ID`	Pointer to the array of bytes.
`int BitCnt`	Number of bits stored in the array.
`DigitCnt`	Number of output digits.
Return:	None.

34.2.12 HostWriteHex

Send a hexadecimal number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteHex(const byte *ID,int BitCnt,int DigitCnt)
```

Parameters:	
`const byte *ID`	Pointer to the array of bytes.
`int BitCnt`	Number of bits stored in the array.
`DigitCnt`	Number of output digits.
Return:	None.

34.2.13 HostWriteVersion

Send the firmware version to the host. This function is sending the result of function `GetVersionString` to the host.

```
void HostWriteVersion(void)
```

Parameters:	None.
Return:	None.

34.3 Beep Functions

Include file: `apptools.h`

The beep functions implement a simplified API around the system function `Beep`.

34.3.1 SetVolume

Set the beeper volume. The default volume is 0.

```
void SetVolume(int NewVolume)
```

Parameters:	
`int NewVolume`	Specify the volume in percent from 0 to 100.
Return:	None.

34.3.2 GetVolume

Read current volume.

```
int GetVolume(void);
```

Parameters:	None.
Return:	Current volume in arange from 0 to 100.

34.3.3 BeepLow

Perform a beep at a frequency of BEEP_FREQUENCY_LOW (2057 Hz) with a duration of 50 milliseconds.

```
void BeepLow(void)
```

Parameters:	None.
Return:	None.

34.3.4 BeepHigh

Perform a beep at a frequency of BEEP_FREQUENCY_HIGH (2400 Hz) with a duration of 50 milliseconds. This is meant to be the standard signal for a successful operation, e.g. read of a transponder.

```
void BeepHigh(void)
```

Parameters:	None.
Return:	None.

34.4 Compatibility to TWN3

Include file: apptools.h

Currently, there is one function for maintaining 100% backward compatibility to TWN3 applications.

34.4.1 ConvertTagTypeToTWN3

This functions converts a tag type from the TWN4 system to TWN3 system. Due to the fact that TWN4 covers a broader range of transponders, the situation might occure, that a conversion is not possible. Under that circumstance the TWN3 value TAGTYPE_NONE (0) is returned.

```
int ConvertTagTypeToTWN3(int TagTypeTWN4)
```

Parameters:	
int TagTypeTWN4	Tag type as returned e.g. by TWN4 system function SearchTag.
Return:	Corresponding tag type as it would be returned by TWN3 system function TagSearch.

34.5 Simple Protocol

Include file: `prs.h`

The Simple Protocol is the standard protocol for building solutions, which need operation of TWN4, which is controlled by the host.

There is a set of functions and definitions, which allow to implement an App, which runs the Simple Protocol. There are some options, which have influence on some details of the Simple Protocol (ASCII/binary mode, CRC). Furthermore, these functions allow to specify custom communication channel and configure the host interface before starting the communication.

The simplest App for using these functions could be written as follows:

```
#include <twn4.sys.h>
#include <prs.h>

int main(void)
{
    InitSimpleProtocol(GetHostChannel(),PRS_COMM_MODE_ASCII | PRS_COMM_CRC_OFF);
    while (true)
    {
        if (SimpleProtoTestCommand())
        {
            SimpleProtoExecuteCommand();
            SimpleProtoSendResponse();
        }
    }
}
```

34.5.1 SimpleProtoInit

Use this function to prepare the Simple Protocol for operation. Before starting this function, it is possible to e.g. prepare a serial port with appropriate communication parameters.

```
bool SimpleProtoInit(int Channel,int Mode)
```

Parameters:	
`int Channel`	This parameter specifies the communication channel for the Simple Protocol. This can be one the channels as defined by the system I/O functions.
`int Mode`	This parameter specified the mode of communication. It is a or-operation, which combines mode (`PRS_COMM_MODE_ASCII` or `PRS_COMM_MODE_BINARY` and CRC (`COMM_CRC_OFF` or `PRS_COMM_CRC_ON`).
Return:	This functions return `true`, if initialization was successful. Otherwise it returns `false`.

34.5.2 SimpleProtoTestCommand

This is a none-blocking function, which polls for the availabilty of a command from the host. If the function returns `true`, a command is available. The command is stored in the global variables `SimpleProtoMessage` and `SimpleProtoMessageLength`.

```
bool SimpleProtoTestCommand(void)
```

Parameters:	None.
Return:	This functions return `true`, if a command became available. Otherwise it returns `false`.

34.5.3 SimpleProtoExecuteCommand

This function executes a command stored in the global variables `SimpleProtoMessage` and `SimpleProtoMessageLengt` After execution of the command, these variables contain the response to be sent to the host.

```
void SimpleProtoExecuteCommand(void)
```

Parameters:	None.
Return:	None.

34.5.4 SimpleProtoSendResponse

This function sends a response stored in the global variables `SimpleProtoMessage` and `SimpleProtoMessageLength` to the host.

```
void SimpleProtoSendResponse(void)
```

Parameters:	None.
Return:	None.

35 Compatibility of TWN4 MultiTech Mini Reader

Due to reduced functionality of TWN4 MultiTech Mini Reader, several API functions are not available. If an API function is called, which is not supported by TWN4 MultiTech Mini Reader the device stops execution of the App and enters exception state (diagnostic LED is flashing three times).

API	Supported	Remark
System Functions	Yes	
I/O Functions	Yes	COM2 is not supported
Memory Functions	Yes	
Peripheral Functions	Yes	Support of GPIO0 to GPIO3 only, Beep is doing delay only
Conversion Functions	Yes	
I2C Functions	No	
RF Functions	Yes	
HITAG 1 and HITAG S Functions	No	
HITAG 2 Functions	No	
EM4x50 Functions	No	
AT55xx Functions	No	
TILF (TIRIS) Functions	No	
LEGIC Functions	No	
MIFARE Classic Functions	Yes	
MIFARE Ultralight (-C) Functions	Yes	
ISO15693 Functions	Yes	
Cryptographic Functions	Yes	
DESFire Functions	Yes	
Contact Card Functions	Yes	SAM1 only
iCLASS Functions	Yes	
ISO14443 Functions	Yes	
NFC SNEP Functions	Yes	
System Parameters	Yes	
Runtime Library	Yes	

36 Disclaimer

Elatec reserves the right to change any information or data in this document without prior notice. The distribution and the update of this document is not controlled. Elatec declines all responsibility for the use of product with any other specifications but the ones mentioned above. Any additional requirement for a specific custom application has to be validated by the customer himself at his own responsibility. Where application information is given, it is only advisory and does not form part of the specification.

All referenced brands, product names, service names and trademarks mentioned in this document are the property of their respective owners.