TWN4

API Reference

DocRev1, February 11, 2013

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 2ˆ32 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_CDC`: HID keyboard

1.7 GetDeviceType

Retrieve type of underlying TWN4 hardware.

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

Parameters:	None.
Return:	`DEVTYPE_LEGICNFC`: TWN4 Legic NFC, `DEVTYPE_MIFARENFC`: TWN4 Mifare NFC

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`.

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);
```

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`

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 Data I/O

2.2.1 Host Communication Channel

2.2.1.1 Send Byte

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.

2.2.1.2 Test Byte

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`.

2.2.1.3 Read Byte

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.

2.2.2 Any Communication Channel

2.2.2.1 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` or `CHANNEL_I2C`.
`int Dir`	Specify the direction. Use one of the predefined constants `DIR_OUT` or `DIR_IN`.
Return:	The buffer size in bytes.

2.2.2.2 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` or `CHANNEL_I2C`.
`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.

2.2.2.3 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` or `CHANNEL_I2C`.
`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`.

2.2.2.4 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` or `CHANNEL_I2C`.
`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`.

2.2.2.5 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` or `CHANNEL_I2C`.
`byte Byte`	The byte to be sent.
Return:	None.

2.2.2.6 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` or `CHANNEL_I2C`.
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

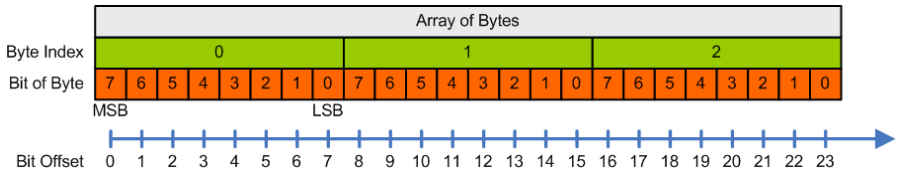
Swap the order of bytes within an array.

```
void FillBytes
    (
    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

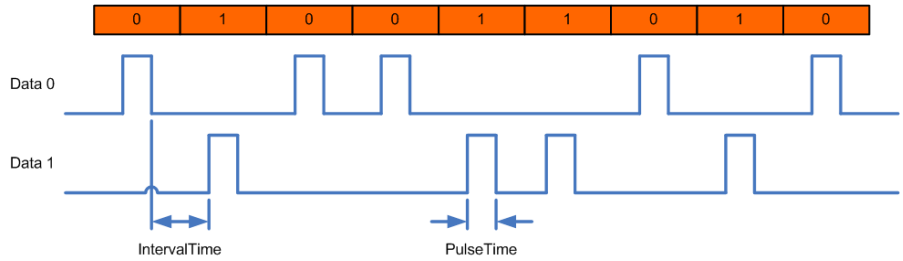
Use this function to send a bitstream via a software emulated Wiegand interface. A Wiegand interface uses two data lines, one line is used to transmit ones, the other one is used to transmit zeros. Each GPIO can be individually configured to act as data line. Note that the integrated API LED-functions 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.

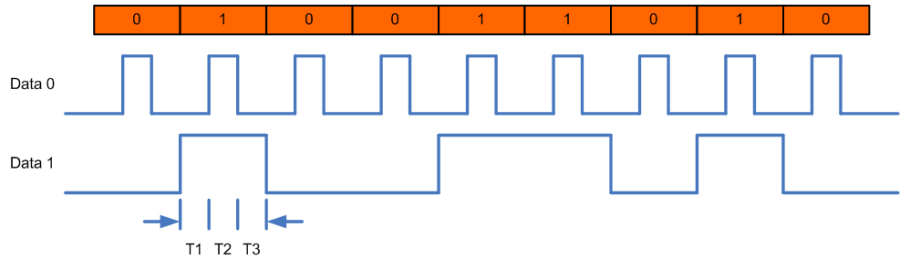
4.1.3.2 Send Data in Omron Format

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 should contain. This allows inhibit of leading digits of an output. If `MaxDigits` is 0, then the number of digits is determined by the given binary data itself.
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:	If the operation was successful, the return value is `true`, otherwise it is `false`.

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);
Byte[0] = I2CMasterReceiveByte();
Byte[1] = I2CMasterReceiveByte();
// Turn off ACK before reading last byte
I2CMasterSetAck(OFF);
Byte[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 RF Functions

7.1 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`.

7.2 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.

7.3 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.

Following transponder types are defined. The table is based on firmware version 1.23.

Definition	Frequency	Name	Supported
LFTAG_EM4102	LF	EM4102	Yes
LFTAG_HITAG1S	LF	Hitag 1 / Hitag S	Yes
LFTAG_HITAG2	LF	Hitag 2	Yes
LFTAG_EM4150	LF	EM4150	On roadmap
LFTAG_AT5555	LF	AT5555 / AT5557 / AT5577 / Q5	Yes
LFTAG_ISOFDX	LF	FDX / EM4105	Yes
LFTAG_EM4026	LF	EM4026	On request
LFTAG_HITAGU	LF	Hitag µ	On request
LFTAG_EM4305	LF	EM4305	Roadmap
LFTAG_HIDPROX	LF	HID Prox	P option only
LFTAG_TIRIS	LF	Tiris / TILF	Yes
LFTAG_COTAG	LF	Cotag	P option only / Hash value
LFTAG_IOPROX	LF	IOProx	P option only
LFTAG_INDITAG	LF	Indala	P option only
LFTAG_HONEYTAG	LF	Honeywell	P option only / Hash value
HFTAG_MIFARE	HF	ID14443A / Mifare	Yes
HFTAG_ISO14443B	HF	ISO14443B	Yes
HFTAG_ISO15693	HF	ISO15693 / Tag-it	Yes
HFTAG_LEGIC	HF	Legic	TWN4 Legic NFC only
HFTAG_HIDICLASS	HF	HID ICLASS	UID only / Full support on roadmap
HFTAG_FELICA	HF	Felica	UID only
HFTAG_SRX	HF	SRC	Yes
HFTAG_NFCP2P	HF	NFC	On roadmap

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));
```

7.4 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.

7.5 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.

8 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	0-255
Hitag S 2048	2048	256	0-255
Hitag S 256	256	32	0-31

8.1 Read/Write Data

8.1.1 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`.

8.1.2 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`.

8.1.3 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`.

8.1.4 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`.

8.2 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`.

9 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.

9.1 Read/Write Data

9.1.1 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`.

9.1.2 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`.

9.1.3 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.

9.2 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`.

10 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.

10.1 Search Function

10.1.1 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`.

10.2 Single-Page Read/Write Function

10.2.1 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`.

10.2.2 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`.

10.2.3 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`.

10.2.4 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`.

10.3 Multi-Page Read/Write Function

10.3.1 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`.

10.3.2 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`.

10.3.3 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`.

10.3.4 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`.

10.3.5 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`.

10.3.6 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`.

10.3.7 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`.

10.3.8 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`.

10.3.9 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`.

10.3.10 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`.

10.3.11 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`.

10.3.12 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`.

10.4 Multi-Usage Read/Write Function

10.4.1 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`.

10.4.2 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`.

10.4.3 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`.

10.4.4 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`.

10.4.5 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`.

10.4.6 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`.

10.4.7 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`.

10.4.8 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`.

10.4.9 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`.

11 Legic-Specific Functions

This chapter describes functions for accessing Legic functionality.

Note:

These functions are available at TWN4 Legic NFC only.

11.1 Direct Access of Legic Chip

TWN4 Legic NFC has a built-in Legic chip type SM4200. There are two 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.

11.1.1 SM4200_GenericRaw

Send a command and receive the response from SM4200. Command and response are expected to include CRC. This function is intended to be used for end-to-end communication between SM4200 and a host.

```
bool SM4200_GenericRaw(const byte *TXData,int TXDataLength,
                       int MaxRXDataLength,int Timeout);
```

Parameters:	
`const byte *TXData`	Pointer to an array of bytes, which contains the command to be sent to SM4200.
`int TXDataLength`	Number of bytes to be sent to SM4200.
`byte *RXData`	Pointer to an array of bytes, which receives response from SM4200
`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 SM4200. This value is specified in milliseconds.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

11.1.2 SM4200_Generic

Send a command and receive the response from SM4200. This function is intended to be used by standand-along applications.

```
bool SM4200_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 SM4200. 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 SM4200. 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 SM4200. This value is specified in milliseconds.
Return:	If the operation was successful, the return value is `true`, otherwise it is `false`.

12 Mifare Classic Specific Transponder Operations

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.

12.1 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 12.1: Well-known keys for Mifare Classic transponders

12.2 Read/Write Data

12.2.1 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`.

12.2.2 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`.

12.3 Handling of Value Blocks

12.3.1 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`.

12.3.2 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`.

12.3.3 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`.

12.3.4 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`.

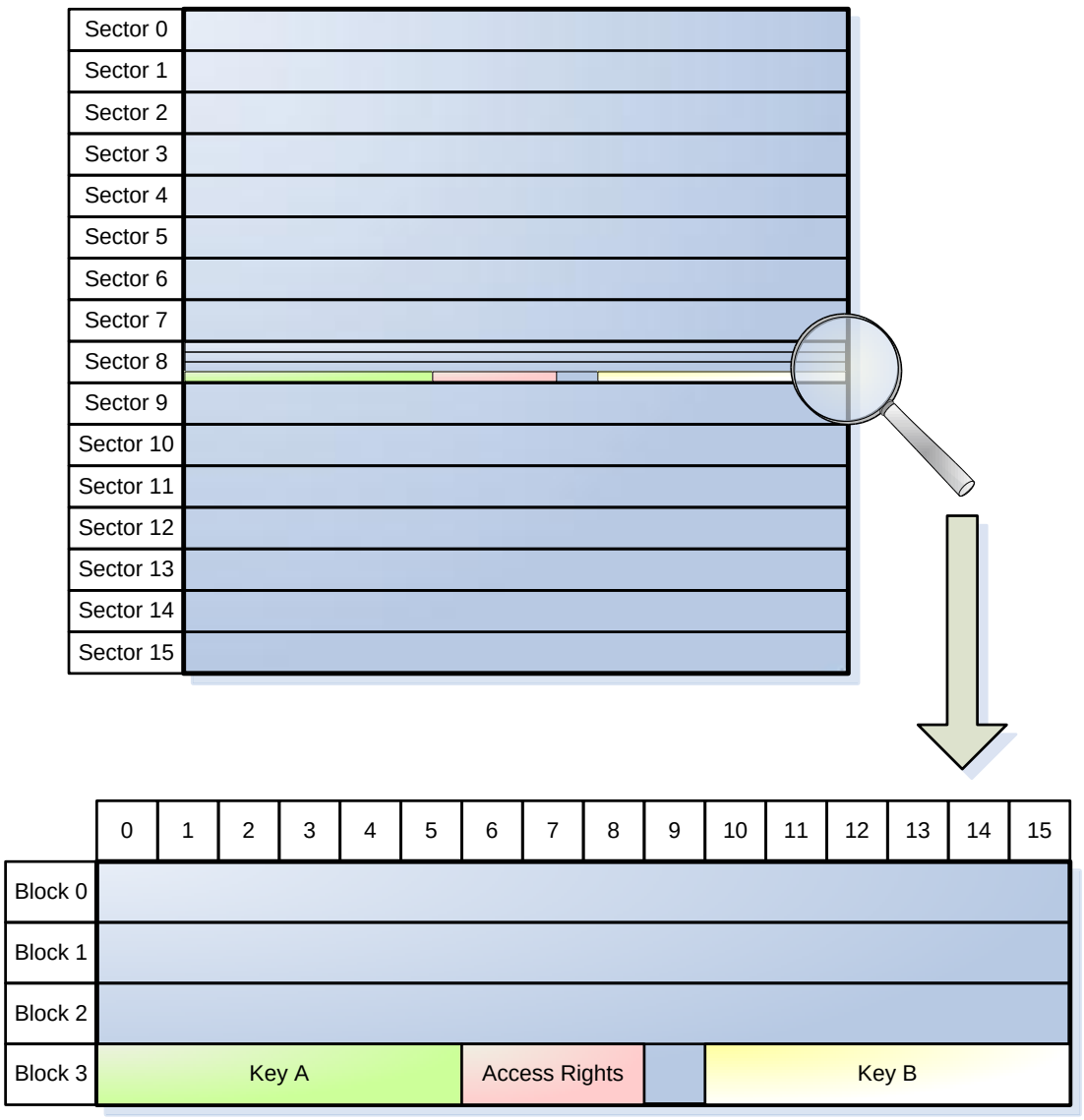


Figure 12.1: Memory layout of a Mifare Classic 1K transponder

13 Mifare Ultralight/Ultralight C Specific Transponder Operations

13.1 Login (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.

```
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 13.1: Well-known key for Mifare Ultralight C transponders

13.2 Read/Write Data

13.2.1 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`.

13.2.2 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`.

14 ISO15693 Specific Transponder Operations

14.1 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`.

14.2 Gather Tag Specific Information

14.2.1 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
    );

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.

14.2.2 Get Tag Type

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

14.2.2.1 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
    );
```

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 14.1: Definition of TISO15693_SystemInfo

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.

Value	Manufacturer	Tag Type
0x00	NXP	ICode SL2
0x01		ICode SL2S
0x0F		Unknown
0x10	TI	Tag-It HFI Plus Inlay
0x11		Tag-It HFI Plus Chip
0x12		Tag-It HFI Standard
0x13		Tag-It HFI Pro
0x1F		Unknown
0x4F	ST	Unknown
0x50	Infineon	SRF55V02P
0x51		SRF55V10P
0x52		SRF55V02S
0x53		SRF55V10S
0x5F		Unknown
0xFF	Unknown	Unknown ISO15693

Table 14.2: Retrievable tag types from UID

14.2.2.2 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.

14.3 Read/Write Data

14.3.1 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`.

14.3.2 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`.

Value	Manufacturer	Tag Type
0x00	NXP	ICode SL2
0x01		ICode SL2S
0x0F		Unknown
0x10	TI	Tag-It HFI Plus Inlay
0x11		Tag-It HFI Plus Chip
0x12		Tag-It HFI Standard
0x13		Tag-It HFI Pro
0x1F		Unknown
0x20	Fuji	MB89R118
0x21		MB89R119
0x2F		Unknown
0x30	ST	24LR16
0x31		24LR64
0x40		LRI1K
0x41		LRI2K
0x42		LRIS2K
0x43		LRIS64K
0x4F		Unknown
0x50	Infineon	SRF55V02P
0x51		SRF55V10P
0x52		SRF55V02S
0x53		SRF55V10S
0x5F		Unknown
0xFF	Unknown	Unknown ISO15693

Table 14.3: Retrievable tag types from System Information

15 Cryptographic Operations

There are two main cryptographic methods available, these are Triple-DES (Data Encryption Standard) and AES (Advanced Encryption Standard). TDES is available in two versions that support different key-lengths: 128 bit (TDES2K) and 192 bit (TDES3K). AES is always based on 128 bit keys.
Each cryptographic method 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 each method provides functions for encryption and decryption.

15.1 Triple-DES

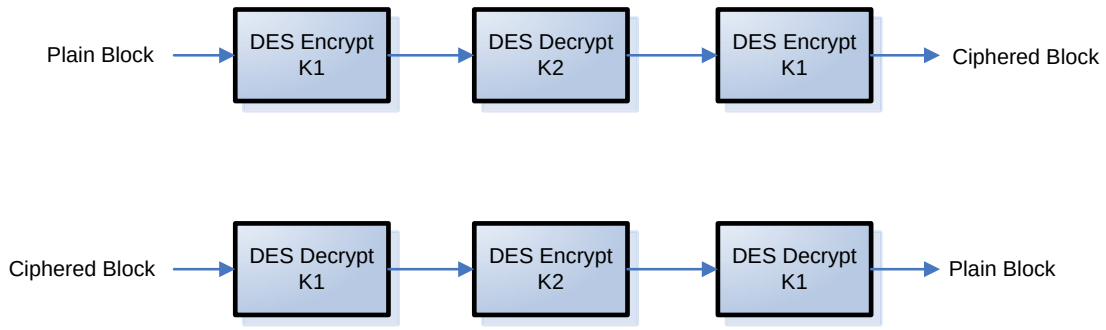


Figure 15.1: TDES2K Operation

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.

15.1.1 Initialization

Use these functions to initialize the TDES2K/3K cryptographic method. During initialization, the 128/192 bit key is passed to the specified cryptographic environment.

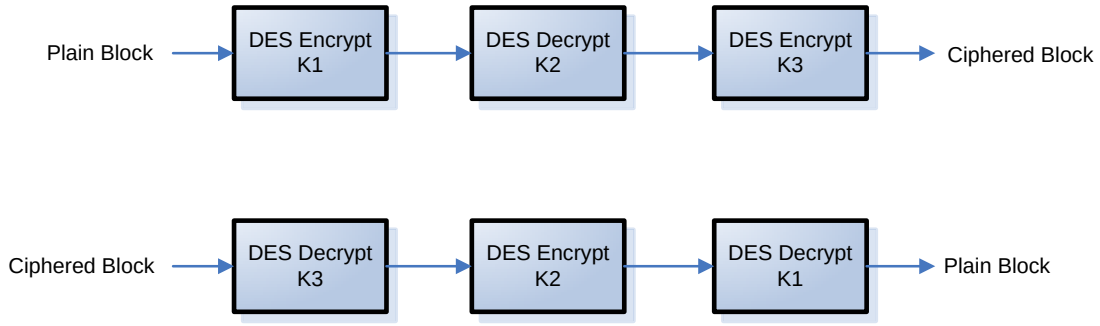


Figure 15.2: TDES3K Operation

```
void TDEA_Init
    (
    int CryptoEnv,
    const byte* Key
    );
void TDES3K_Init
    (
    int CryptoEnv,
    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.
`const byte* Key`	The key is passed by this parameter. Depending on the init-function, the key-length is either 16 or 24 bytes.
Return:	This function has no return value.

15.1.2 Encrypt

Use these functions to encrypt 8 bytes of plain data.

```
void TDEA_Encrypt
    (
    int CryptoEnv,
    const byte* Plain,
    byte* Cipher
    );
```

```
void TDES3K_Encrypt
    (
    int CryptoEnv,
    const byte* Plain,
    byte* Cipher
    );
```

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. The environment must have been initialized before it can be used.
`const byte* Plain`	Pointer to the array, that contains the plain data block to be encrypted.
`byte* Cipher`	Pointer to the array, that receives the encrypted data block. TDES always operates on 8 byte blocks, so take care for proper dimensioning of the array.
Return:	This function has no return value.

15.1.3 Decrypt

Use these functions to decrypt a 8 bytes data block.

```
void TDEA_Decrypt
    (
    int CryptoEnv,
    const byte* Cipher,
    byte* Plain
    );
void TDES3K_Decrypt
    (
    int CryptoEnv,
    const byte* Cipher,
    byte* Plain
    );
```

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. The environment must have been initialized before it can be used.
`const byte* Cipher`	Pointer to the array, that contains the ciphered data block to be decrypted.
`byte* Plain`	Pointer to the array, that receives the decrypted data block. TDES always operates on 8 byte blocks, so take care for proper dimensioning of the array.
Return:	This function has no return value.

15.2 AES

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.

15.2.1 Initialization

Use this function to initialize the AES cryptographic method. During initialization, the 128 bit key is passed to the specified cryptographic environment.

```
void AES128_Init
    (
    int CryptoEnv,
    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.
`const byte* Key`	The key is passed by this parameter. The key-length is 16 bytes.
Return:	This function has no return value.

15.2.2 Encrypt

Use this function to encrypt 16 bytes of plain data.

```
void AES128_Enc
    (
    int CryptoEnv,
    const byte* Plain,
    byte* Cipher
    );
```

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. The environment must have been initialized before it can be used.
`const byte* Plain`	Pointer to the array, that contains the plain data block to be encrypted.
`byte* Cipher`	Pointer to the array, that receives the encrypted data block. AES always operates on 16 byte blocks, so take care for proper dimensioning of the array.
Return:	This function has no return value.

15.2.3 Decrypt

Use this function to decrypt a 16 bytes data block.

```
void AES128_Dec
    (
    int CryptoEnv,
    const byte* Cipher,
    byte* Plain
    );
```

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. The environment must have been initialized before it can be used.
`const byte* Cipher`	Pointer to the array, that contains the ciphered data block to be decrypted.
`byte* Plain`	Pointer to the array, that receives the decrypted data block. AES always operates on 16 byte blocks, so take care for proper dimensioning of the array.
Return:	This function has no return value.

16 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. A simple use-case could be: Search for a transpon-

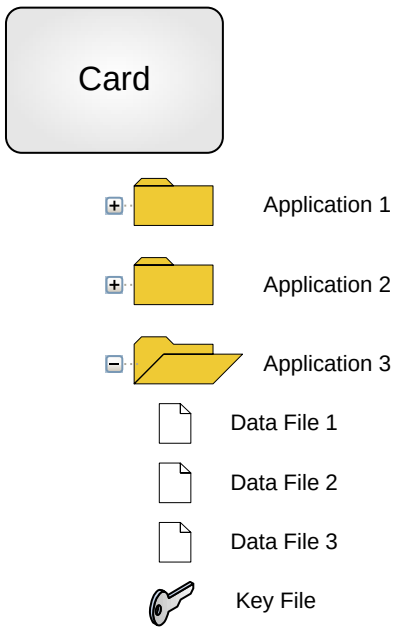


Figure 16.1: DESFire memory layout

der, select the desired application, perform an authentication with the respective key (if required), access data file for read or write operation.

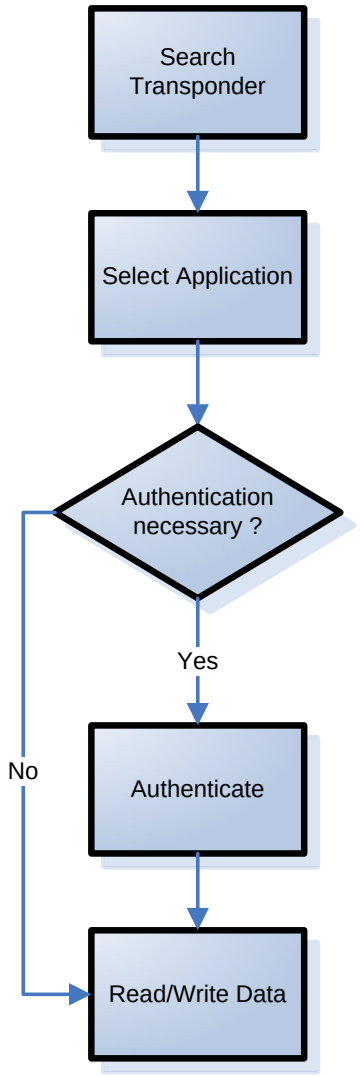


Figure 16.2: Simple way to gain access to the file system

16.1 Security Related Operations

16.1.1 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 cryp-

tographic 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();
}
```

16.1.2 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();
}
```

16.1.3 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`.

Example:

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 16.1: Definition of `TDESFireMasterKeySettings`

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

TDESFireMasterKeySettings MasterKeySettings;
```

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 16.2: Definition of `TDESFireKeySettings`

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

```
        DoSomething(MasterKeySettings);
    }
}
```

16.1.4 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`.

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

16.2 Transponder Related Operations

16.2.1 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();
}
```

16.2.2 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`.

16.2.3 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);
}
```

16.2.4 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`.

16.2.5 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
    );
```

16.2.6 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`	32	This member holds manufacturing specific information.

Table 16.3: 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 16.4: 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 16.5: Definition of `TDESFireProdInfo`

16.2.7 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`.

16.2.8 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`.

16.2.9 Set Transponder Configuration

16.2.9.1 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`.

16.2.9.2 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`.

16.2.9.3 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`.

16.2.9.4 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_SetDefaultKey
    (
    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`.

16.3 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:

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 16.6: Definition of `TDESFireFileSettings`

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`.

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

Table 16.7: Coding of `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.

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 16.8: Definition of `union TDESFireSpecificFileInfo`

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

Table 16.9: Definition of `struct TDESFireDataFileSettings`

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.
`bool LimitedCreditEnabled`	32	LimitedCredit feature enabled or disabled.

Table 16.10: Definition of `struct TDESFireValueFileSettings`

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 16.11: Definition of `struct TDESFireRecordFileSettings`

16.3.1 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 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.

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

bool DESFire_CreateValueFile
    (
    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();
    }
}
```

16.3.2 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`.

16.3.3 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.

16.3.4 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;
                }
            }
        }
    }
}
```

16.3.5 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`.

16.4 File Related Operations

16.4.1 Data Files

16.4.1.1 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();
}
```

16.4.1.2 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();
}
```

16.4.2 Value Files

16.4.2.1 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`.

16.4.2.2 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.

16.4.2.3 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.

16.4.2.4 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.

16.4.3 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`.

16.4.4 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`.