

# TWN4

## API Reference

DocRev8, September 12, 2014



Elatec GmbH

# Contents

1	System Functions	10
1.1	SysCall	10
1.2	Reset	10
1.3	StartBootloader	10
1.4	GetSysTicks	11
1.5	GetVersionString	11
1.6	GetUSBType	12
1.7	GetDeviceType	12
1.8	Sleep	12
1.9	GetDeviceUID	13
1.10	SetParameters	13
1.11	GetLastError	14
2	I/O Functions	15
2.1	Configuration	15
2.1.1	Set COM-Port Parameters	15
2.1.2	Get USB Device State	15
2.1.3	Get Host Channel	16
2.2	Data I/O	17
2.2.1	Query I/O Buffer Size	17
2.2.2	Get I/O Buffer Byte Count	17
2.2.3	Test Empty	18
2.2.4	Test Full	18
2.2.5	Send Byte	19
2.2.6	Read Byte	19
3	Memory Functions	20
3.1	Byte Operations	20
3.1.1	Compare Bytes	20
3.1.2	Copy Bytes	20
3.1.3	Fill Bytes	21
3.1.4	Swap Bytes	21
3.2	Bit Operations	22
3.2.1	Read Bit	22
3.2.2	Write Bit	22
3.2.3	Copy Bit	23
3.2.4	Compare Bits	24
3.2.5	Copy Bits	24

3.2.6	Fill Bits . . . . .	25
3.2.7	Swap Bits . . . . .	25
3.2.8	Count Bits . . . . .	26
4	Peripheral Functions . . . . .	27
4.1	General Purpose Inputs/Outputs (GPIOs) . . . . .	27
4.1.1	Configuration . . . . .	27
4.1.1.1	Outputs . . . . .	27
4.1.1.2	Inputs . . . . .	28
4.1.2	Basic Port Functions . . . . .	28
4.1.2.1	Set GPIOs to Logical Level . . . . .	28
4.1.2.2	Toggle GPIOs . . . . .	29
4.1.2.3	Waveform Generation . . . . .	29
4.1.2.4	Read GPIOs . . . . .	30
4.1.3	Higher Level Port Functions . . . . .	30
4.1.3.1	Send Data in Wiegand Format . . . . .	30
4.1.3.2	Send Data in Omron Format . . . . .	32
4.2	Beeper . . . . .	33
4.3	LEDs . . . . .	34
4.3.1	General Purpose LED Functions . . . . .	34
4.3.1.1	Initialization . . . . .	34
4.3.1.2	Set LEDs On/Off . . . . .	35
4.3.1.3	Toggle LEDs . . . . .	35
4.3.1.4	Blink LEDs . . . . .	35
4.3.1.5	Get LED State . . . . .	36
4.3.2	Diagnostic LED . . . . .	36
4.3.2.1	Set Diagnostic LED On/Off . . . . .	36
4.3.2.2	Toggle Diagnostic LED . . . . .	37
4.3.2.3	Get LED State . . . . .	37
5	Conversion Functions . . . . .	38
5.1	Hexadecimal ASCII to Binary . . . . .	38
5.1.1	Scan Hexadecimal Character . . . . .	38
5.1.2	Scan Hexadecimal String . . . . .	38
5.2	Binary to Hexadecimal ASCII . . . . .	39
6	I2C Functions . . . . .	41
6.1	Initialization/Deinitialization . . . . .	41
6.1.1	I2CInit . . . . .	41
6.1.2	I2CDeInit . . . . .	41
6.1.3	Examples . . . . .	41
6.2	Communication (Master) . . . . .	42
6.2.1	I2CMasterStart . . . . .	42
6.2.2	I2CMasterStop . . . . .	42
6.2.3	I2CMasterTransmitByte . . . . .	42
6.2.4	I2CMasterReceiveByte . . . . .	42
6.2.5	I2CMasterBeginWrite . . . . .	43

6.2.6	I2CMasterBeginRead . . . . .	43
6.2.7	I2CMasterSetAck . . . . .	43
6.2.8	Examples . . . . .	43
6.3	Communication (Slave) . . . . .	44
6.3.1	Slave to Master . . . . .	45
6.3.2	Master to Slave . . . . .	45
6.3.3	Examples . . . . .	45
7	RF Functions . . . . .	47
7.1	SearchTag . . . . .	47
7.2	SetRFOff . . . . .	47
7.3	SetTagTypes . . . . .	48
7.3.1	Supported Types of LF Tags (125 kHz - 134.2 kHz) . . . . .	49
7.3.2	Supported Types of HF Tags (13.56 MHz) . . . . .	50
7.4	GetTagTypes . . . . .	50
7.5	GetSupportedTagTypes . . . . .	51
8	HITAG 1- and HITAG S-Specific Transponder Operations . . . . .	52
8.1	Read/Write Data . . . . .	52
8.1.1	Hitag1S_ReadPage . . . . .	52
8.1.2	Hitag1S_WritePage . . . . .	53
8.1.3	Hitag1S_ReadBlock . . . . .	53
8.1.4	Hitag1S_WriteBlock . . . . .	53
8.2	Hitag1S_Halt . . . . .	54
9	HITAG 2-Specific Transponder Operations . . . . .	55
9.1	Read/Write Data . . . . .	55
9.1.1	Hitag2_ReadPage . . . . .	55
9.1.2	Hitag2_WritePage . . . . .	55
9.1.3	Hitag2_SetPassword . . . . .	56
9.2	Hitag2_Halt . . . . .	56
10	EM4x50-Specific Transponder Operations . . . . .	57
10.1	Functions . . . . .	57
10.1.1	EM4150_Login . . . . .	57
10.1.2	EM4150_ReadWord . . . . .	57
10.1.3	EM4150_WriteWord . . . . .	58
10.1.4	EM4150_WritePassword . . . . .	58
11	AT55xx-Specific Transponder Operations . . . . .	59
11.1	Control Functions . . . . .	59
11.1.1	AT55_Begin . . . . .	59
11.2	Read Data . . . . .	59
11.2.1	AT55_ReadBlock . . . . .	60
11.2.2	AT55_ReadBlockProtected . . . . .	60
11.3	Write Data . . . . .	61
11.3.1	AT55_WriteBlock . . . . .	61
11.3.2	AT55_WriteBlockProtected . . . . .	61
11.3.3	AT55_WriteBlockAndLock . . . . .	62

11.3.4	AT55_WriteBlockProtectedAndLock	62
12	TILF (TIRIS) Functions	63
12.1	Search Function	63
12.1.1	TILF_SearchTag	63
12.2	Single-Page Read/Write Function	64
12.2.1	TILF_ChargeOnlyRead	64
12.2.2	TILF_ChargeOnlyReadLo	64
12.2.3	TILF_SPPProgramPage	64
12.2.4	TILF_SPPProgramPageLo	65
12.3	Multi-Page Read/Write Function	65
12.3.1	TILF_MPGeneralReadPage	65
12.3.2	TILF_MPSelectiveReadPage	65
12.3.3	TILF_MPPProgramPage	66
12.3.4	TILF_MPSelectiveProgramPage	66
12.3.5	TILF_MPLockPage	67
12.3.6	TILF_MPSelectiveLockPage	67
12.3.7	TILF_MPGeneralReadPageLo	68
12.3.8	TILF_MPSelectiveReadPageLo	68
12.3.9	TILF_MPPProgramPageLo	68
12.3.10	TILF_MPSelectiveProgramPageLo	69
12.3.11	TILF_MPLockPageLo	69
12.3.12	TILF_MPSelectiveLockPageLo	70
12.4	Multi-Usage Read/Write Function	70
12.4.1	TILF_MUGeneralReadPage	70
12.4.2	TILF_MUSelectiveReadPage	70
12.4.3	TILF_MUSpecialReadPage	71
12.4.4	TILF_MUProgramPage	71
12.4.5	TILF_MUSelectiveProgramPage	72
12.4.6	TILF_MUSpecialProgramPage	72
12.4.7	TILF_MULockPage	72
12.4.8	TILF_MUSelectiveLockPage	73
12.4.9	TILF_MUSpecialLockPage	73
13	ISO14443 Transponder Operations	74
13.1	ISO14443A	74
13.1.1	Get ATQA	74
13.1.2	Get SAK	74
13.1.3	Get ATS	75
13.2	ISO14443B	75
13.2.1	Get ATQB	75
13.2.2	Get Answer to ATTRIB	76
13.3	Check Presence	76
13.4	ISO14443-4 Transparent Data Exchange	77
14	MIFARE Classic Specific Transponder Operations	78
14.1	Login	78

14.2	Read/Write Data	79
14.2.1	Read Data Block	79
14.2.2	Write Data Block	80
14.3	Handling of Value Blocks	80
14.3.1	Read Value Block	80
14.3.2	Write Value Block	81
14.3.3	Increment Value Block	81
14.3.4	Decrement Value Block	82
15	MIFARE Ultralight/Ultralight C Specific Transponder Operations	84
15.1	Login (Ultralight C only)	84
15.2	Read/Write Data	85
15.2.1	Read Page	85
15.2.2	Write Page	85
16	DESFire Specific Transponder Operations	87
16.1	Security Related Operations	88
16.1.1	Authenticate	88
16.1.2	Get Key Version	91
16.1.3	Get Key Settings	92
16.1.4	Change Key Settings	94
16.1.5	Change Key	94
16.2	Transponder Related Operations	96
16.2.1	Create Application	96
16.2.2	Delete Application	97
16.2.3	Get Application IDs	98
16.2.4	Select Application	99
16.2.5	Format Transponder	100
16.2.6	Get Transponder Information	100
16.2.7	Get Available Memory Space	102
16.2.8	Get Card UID	102
16.2.9	Set Transponder Configuration	103
16.2.9.1	Disable Format Tag	103
16.2.9.2	Enable Random ID	104
16.2.9.3	Set Default Key	104
16.2.9.4	Set User-defined Answer To Select (ATS)	105
16.3	Application Related Operations	106
16.3.1	Create File	108
16.3.2	Delete File	110
16.3.3	Get File IDs	110
16.3.4	Get File Settings	111
16.3.5	Change File Settings	113
16.4	File Related Operations	114
16.4.1	Data Files	114
16.4.1.1	Read Data	114
16.4.1.2	Write Data	116

16.4.2	Value Files . . . . .	119
16.4.2.1	Get Value . . . . .	119
16.4.2.2	Debit . . . . .	120
16.4.2.3	Credit . . . . .	121
16.4.2.4	Limited Credit . . . . .	122
16.4.3	Commit Transaction . . . . .	123
16.4.4	Abort Transaction . . . . .	124
17	ISO15693 Specific Transponder Operations . . . . .	125
17.1	Generic ISO15693 Command . . . . .	125
17.2	Gather Tag Specific Information . . . . .	126
17.2.1	Get System Information . . . . .	126
17.2.2	Get Tag Type . . . . .	127
17.2.2.1	Get Tag Type From UID . . . . .	127
17.2.2.2	Get Tag Type From System Information . . . . .	129
17.3	Read/Write Data . . . . .	131
17.3.1	Read Single Block . . . . .	131
17.3.2	Write Single Block . . . . .	131
18	LEGIC-Specific Functions . . . . .	133
18.1	Direct Access of LEGIC Chip . . . . .	133
18.1.1	SM4X00_GenericRaw . . . . .	133
18.1.2	SM4X00_Generic . . . . .	134
18.1.3	SM4X00_StartBootloader . . . . .	135
18.1.4	SM4X00_EraseFlash . . . . .	135
18.1.5	SM4X00_ProgramBlock . . . . .	136
19	iCLASS Specific Transponder Operations . . . . .	137
19.1	Read PAC Bits . . . . .	137
19.2	Example . . . . .	138
20	Simple NDEF Exchange Protocol (SNEP) . . . . .	139
20.1	Initialize SNEP Service . . . . .	139
20.2	Get Connection State . . . . .	140
20.3	Query Message FIFO . . . . .	140
20.4	Transmit NDEF Message . . . . .	141
20.4.1	Begin Message . . . . .	141
20.4.2	Send Message Fragment . . . . .	141
20.4.3	Example . . . . .	143
20.5	Receive NDEF Message . . . . .	145
20.5.1	Test Message . . . . .	145
20.5.2	Receive Message Fragment . . . . .	145
20.5.3	Example . . . . .	147
21	Contact Card Operations . . . . .	149
21.1	Query Card Slot Status . . . . .	149
21.2	Card Activation . . . . .	151
21.3	Card Deactivation . . . . .	152
21.4	Set Communication Settings . . . . .	153

21.5	Transparent Data Transmission	155
21.6	Exchange Of APDUs	157
21.7	Examples	159
21.7.1	PPS Example	159
21.7.2	Communication Example	160
22	Cryptographic Operations	162
22.1	Initialization	165
22.2	Encrypt	166
22.3	Decrypt	167
22.4	Reset Init Vector	167
23	Storage Functions	168
23.1	Management Functions	169
23.1.1	FSMount	169
23.1.2	FSFormat	169
23.2	File Functions	170
23.2.1	FSOpen	170
23.2.2	FSClose	171
23.2.3	FSCloseAll	171
23.2.4	FSSeek	172
23.2.5	FSTell	172
23.2.6	FSReadBytes	173
23.2.7	FSWriteBytes	173
23.3	Directory Functions	174
23.3.1	FSFindFirst	174
23.3.2	FSFindNext	175
23.3.3	FSDelete	175
23.3.4	FSRename	175
23.3.5	Examples	176
24	System Parameters	179
24.1	TLV Format	179
24.2	Manifest	179
24.3	Available Parameters	181
25	System Errors	183
26	Runtime Library	185
26.1	Timer Functions	185
26.1.1	StartTimer	185
26.1.2	StopTimer	185
26.1.3	TestTimer	186
26.2	Host Communication	186
26.2.1	SetHostChannel	186
26.2.2	HostTestByte	187
26.2.3	HostReadByte	187
26.2.4	HostTestChar	187
26.2.5	HostReadChar	187



26.2.6	HostWriteByte . . . . .	188
26.2.7	HostWriteChar . . . . .	188
26.2.8	HostWriteString . . . . .	188
26.2.9	HostWriteRadix . . . . .	188
26.2.10	HostWriteBin . . . . .	189
26.2.11	HostWriteDec . . . . .	189
26.2.12	HostWriteHex . . . . .	190
26.2.13	HostWriteVersion . . . . .	190
26.3	Beep Functions . . . . .	190
26.3.1	SetVolume . . . . .	190
26.3.2	GetVolume . . . . .	191
26.3.3	BeepLow . . . . .	191
26.3.4	BeepHigh . . . . .	191
26.4	Compatibility to TWN3 . . . . .	191
26.4.1	ConvertTagTypeToTWN3 . . . . .	191
26.5	Simple Protocol . . . . .	192
26.5.1	SimpleProtoInit . . . . .	193
26.5.2	SimpleProtoTestCommand . . . . .	193
26.5.3	SimpleProtoExecuteCommand . . . . .	193
26.5.4	SimpleProtoSendResponse . . . . .	194
27	Compatibility of TWN4 Mini Reader MIFARE NFC . . . . .	195

# 1 System Functions

## 1.1 SysCall

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

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

Parameters:

TEnvSysCall *Env	Pointer to a structure which specifies parameters of the functions to be called.
------------------	--

Return:

If the function has been called the return value is `true`, otherwise it is `false`. In this case the specified function does not exist.

## 1.2 Reset

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

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

Parameters:

None.

Return:

None.

## 1.3 StartBootloader

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

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

Parameters:

None.

Return:

None.

## 1.4 GetSysTicks

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

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

Parameters: None.

Return: Number of system ticks since startup of the firmware. The returned value will restart at 0 after  $2^{32}$  system ticks (around 1193 hours).

## 1.5 GetVersionString

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

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

Parameters:

`char *VersionString` Pointer to an array of characters, which will receive the version information.

`int MaxLen` Maximum number of characters, the specified byte array can receive excluding the 0-termination.

Return: Length of the returned string excluding the 0-termination.

Example:

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

## 1.6 GetUSBType

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

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

<u>Parameters:</u>	None.
<u>Return:</u>	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);
```

<u>Parameters:</u>	None.
<u>Return:</u>	DEVTYPE_LEGICNFC: TWN4 LEGIC NFC, DEVTYPE_MIFARENFC: TWN4 MIFARE NFC

## 1.8 Sleep

The device enters the sleep state for a specified time. During sleep state, the device reduces the current consumption to as low as 8mA (TWN4 MIFARE NFC) and 20mA (TWN4 LEGIC NFC).

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

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

The sleep state can optionally be interrupted by events. These events are specified as parameters in the call of the function Sleep. There are several definitions for the their corresponding events.

Definition	Value	Description
WAKEUP_BY_USB_MSK	0x01	The USB input channel received at least on byte.
WAKEUP_BY_COM1_MSK	0x02	The input channel of COM1 received at least on byte.
WAKEUP_BY_COM2_MSK	0x04	The input channel of COM2 received at least on byte.

## 1.9 GetDeviceUID

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

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

### Parameters:

**byte \*UID** Pointer to an array of bytes, which receives 12 bytes. These 12 bytes represent the UID of the device.

**Return:** None.

## 1.10 SetParameters

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

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

### Parameters:

**const byte \*TLV** Pointer to an array of bytes, which contains the TLV list.

**int ByteCount** Length counted in bytes, the TLV list contains.

**Return:** The function returns `true`, if the parameters was set to the new value. Otherwise the function returns `false`.

Example:

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

int main(void)
```

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

## 1.11 GetLastError

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

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

<u>Parameters:</u>	None.
<u>Return:</u>	The error code.

## 2 I/O Functions

### 2.1 Configuration

#### 2.1.1 Set COM-Port Parameters

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

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

Parameters:

`int` Channel

Specify the communication port which shall be configured. Use one of the predefined constants CHANNEL\_COM1 or CHANNEL\_COM2.

TCOMParameters\*  
COMParameters

Reference to the structure that holds the communication parameters. See the description of TCOMParameters for details.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`.

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

Table 2.1: Definition of TCOMPParameters

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 Query I/O Buffer Size

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

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

Parameters:

`int Channel` Specify the communication channel. Use one of the predefined constants CHANNEL\_USB, CHANNEL\_COM1, CHANNEL\_COM2, CHANNEL\_I2C or CHANNEL\_RNG.

`int Dir` Specify the direction. Use one of the predefined constants DIR\_OUT or DIR\_IN.

Return: The buffer size in bytes.

### 2.2.2 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, CHANNEL\_I2C or CHANNEL\_RNG.

`int Dir` Specify the direction. Use one of the predefined constants DIR\_OUT or DIR\_IN.

Return: The number of bytes that are stored in the buffer.

### 2.2.3 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, CHANNEL\_I2C or CHANNEL\_RNG.

`int` Dir

Specify the direction. Use one of the predefined constants DIR\_OUT or DIR\_IN.

Return:

If the buffer is empty, the return value is `true`, otherwise it is `false`.

### 2.2.4 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, CHANNEL\_I2C or CHANNEL\_RNG.

`int` Dir

Specify the direction. Use one of the predefined constants DIR\_OUT or DIR\_IN.

Return:

If the buffer is full, the return value is `true`, otherwise it is `false`.

### 2.2.5 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.

### 2.2.6 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, CHANNEL\_I2C or CHANNEL\_RNG.

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:

<code>const byte* Data1</code>	Reference to an array of bytes.
<code>const byte* Data2</code>	Reference to an array of bytes.
<code>int ByteCount</code>	Number of bytes (beginning from index 0) to be compared.
<u>Return:</u>	If the two arrays are identical, the return value is <code>true</code> , otherwise it is <code>false</code> .

#### 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:

<code>byte*</code> DestBytes	Reference to an array of bytes which is the destination of the copy operation.
<code>const byte*</code> SourceBytes	Reference to an array of bytes which is the source of the copy operation.
<code>int</code> ByteCount	Number of bytes to be copied.
<u>Return:</u>	None.

### 3.1.3 Fill Bytes

Fill bytes within a given array with a value.

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

Parameters:

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

### 3.1.4 Swap Bytes

Swap the order of bytes within an array.

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

Parameters:

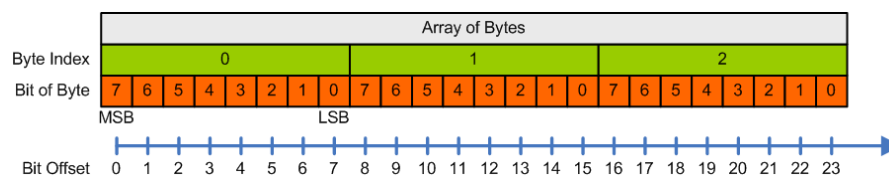
`byte*` Data                      Reference to an array of bytes which is the destination for the operation.

`int` ByteCount                      Number of bytes to be swapped.

Return:                              None.

## 3.2 Bit Operations

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



### 3.2.1 Read Bit

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

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

Parameters:

`const byte*` Byte                      Reference to an array of bytes which represents the bit field where one bit shall be read.

`int` BitNr                              Position of the bit within the bit field.

Return:                              The bit value: true means 1, false means 0.

### 3.2.2 Write Bit

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

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

**Parameters:**

<code>byte*</code> Byte	Reference to an array of bytes which represents the bit field where one bit shall be written.
<code>int</code> BitNr	Position within the bit field, where the bit is to be written.
<code>bool</code> Value	The bit value: true means 1, false means 0.
<u>Return:</u>	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:**

<code>byte*</code> Dest	Reference to an array of bytes which is the destination for the operation.
<code>int</code> DestBitNr	Position within the destination bit field, where the bit is copied to.
<code>const byte*</code> Source	Reference to an array of bytes which is the source for the operation.
<code>int</code> SourceBitNr	Position within the source bit field, where the bit is copied from.
<u>Return:</u>	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:

<code>const byte* Data1</code>	Reference to an array of bytes which represents a bit field.
<code>int Data1StartBit</code>	Start-index (beginning from 0) of the first bit field.
<code>const byte* Data2</code>	Reference to an array of bytes which represents a bit field.
<code>int Data2StartBit</code>	Start-index (beginning from 0) of the second bit field.
<code>int BitCount</code>	Number of bits to be compared.
<u>Return:</u>	If the two bit-sets are identical, the return value is <code>true</code> , otherwise it is <code>false</code> .

### 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:

<code>byte*</code> DestBits	Reference to an array of bytes which represents a bit field which is the destination of the copy operation.
<code>int</code> StartDestBit	First bit within the destination bit field where the bits are copied to.
<code>const byte*</code> SourceBits	Reference to an array of bytes which represents a bit field which is the source of the copy operation.
<code>int</code> StartSourceBit	First bit within the source bit field where the bits are copied from.
<code>int</code> 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:

<code>byte*</code> Dest	Reference to an array of bytes which represents a bit field which is the destination for the operation.
<code>int</code> StartBit	First bit within the bit field where the bits are filled.
<code>bool</code> Value	The bit value: <code>true</code> means 1, <code>false</code> means 0.
<code>int</code> 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:**

<code>byte*</code> Data	Reference to an array of bytes which represents a bit field which is the destination for the operation.
<code>int</code> StartBit	First bit within the bit field where bits are swapped.
<code>int</code> BitCount	Number of bits to be swapped.
<u>Return:</u>	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:**

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

<code>int</code> 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.
<code>int</code> 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.
<code>int</code> OutputType	Specify the output driver characteristic. Use one of the predefined constants GPIO_OTYPE_PUSH_PULL 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:

<code>int</code> 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.
<code>int</code> 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:

<code>int</code> 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 GPIONToggleBits
(
    int Bits
);
```

Parameters:

<code>int</code> 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.
-----------------------	---

<u>Return:</u>	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:

<code>int</code> 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.
-----------------------	--

<code>int</code> TimeHi	Specify the duration for logical high level in milliseconds.
-------------------------	--

<code>int</code> TimeLo	Specify the duration for logical low level in milliseconds.
-------------------------	---

<u>Return:</u>	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 GPIONTestBit
(
    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.

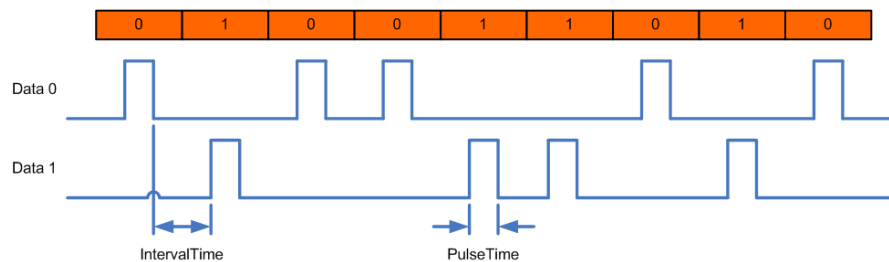
```
void SendWiegand(int GPIOData0,int GPIOData1,int PulseTime,
                int IntervalTime,byte* Bits,int BitCount);
```

Parameters:

<code>int</code> <code>GPIOData0</code>	Specify the GPIO that shall be used to transmit zeros. Use one of the predefined constants <code>GPIO0</code> through <code>GPIO7</code> for specifying the GPIO.
<code>int</code> <code>GPIOData1</code>	Specify the GPIO that shall be used to transmit ones. Use one of the predefined constants <code>GPIO0</code> through <code>GPIO7</code> for specifying the GPIO.
<code>int</code> <code>PulseTime</code>	Specify the pulse duration in microseconds.
<code>int</code> <code>IntervalTime</code>	Specify the duration in microseconds between consecutive pulses.
<code>byte*</code> <code>Bits</code>	Reference to an array of bytes which represents a bit field which holds the data to be sent.
<code>int</code> <code>BitCount</code>	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);
```

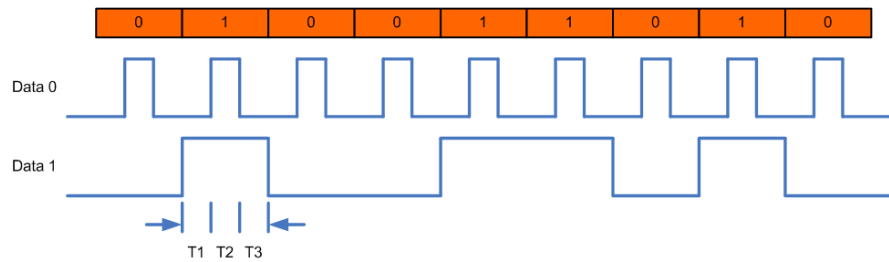
##### Parameters:

<code>int GPIOClock</code>	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.
<code>int GPIOData</code>	Specify the GPIO that shall be used for data transmission. Use one of the predefined constants GPIO0 through GPIO7 for specifying the GPIO.
<code>int T1</code>	
<code>int T2</code>	
<code>int T3</code>	
<code>byte* Bits</code>	Reference to an array of bytes which represents a bit field which holds the data to be sent.
<code>int BitCount</code>	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_PUSH_PULL);
// 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.
<u>Return:</u>	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.
<u>Return:</u>	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);
```

<u>Parameters:</u>	None.
<u>Return:</u>	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:

<code>byte Char</code>	ASCII-coded hexadecimal character. The input value may be one of the characters '0'-'9', 'a'-'f' or 'A'-'F'.
------------------------	--

<u>Return:</u>	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:

<code>const byte*</code> SourceBits	A reference to an array of bytes, which contains the bit field.
<code>int</code> StartBit	Index of the first bit to be converted.
<code>int</code> BitCnt	The number of bits, which are valid within the array of bytes.
<code>char*</code> String	A reference to an array of bytes, which receives the result of the conversion.
<code>int</code> Radix	Base for conversion, use: <ul style="list-style-type: none"><li>• 2 for binary conversion</li><li>• 8 for octal conversion</li><li>• 10 for decimal conversion</li><li>• 16 for hexadecimal conversion</li></ul>
<code>int</code> 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.
<code>int</code> 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 reception 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 reception of data
// to/from a I2C-slave

// This is the address of the slave
```

```
const int I2CAddress = 0x30;
// Init the I2C port
I2CInit(I2CMODE_MASTER);

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

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

### 6.3 Communication (Slave)

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

The communication is performed via normal communication channels. Therefore, for transmitting and receiving data, the normal IO-functions must be used. These are WriteByte, ReadByte and so on. In case of communication via I2C, the channel 4 must be used. There is a definition for this channel, which is CHANNEL\_I2C.

As a conclusion, TWN4 offers a easy method of changing communication from USB or RS232 to I2C just by changing the communication channel. Only care must be taken to avoid buffer overflow. This can be achieved by calling appropriate IO-functions TestEmpty and TestFull. On the other hand many communication protocols avoid a buffer overflow by their inherent flow of communication (e.g. command/response protocol).

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

### 6.3.1 Slave to Master

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

### 6.3.2 Master to Slave

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

### 6.3.3 Examples

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

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

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

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

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

        I2CMasterStart();
```

```

I2CMasterBeginRead(I2CAddress);
I2CMasterSetAck(OFF);
I2CRXTXCount = I2CMasterReceiveByte();
I2CMasterStop();

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

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

```

## 7 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);
```

<u>Parameters:</u>	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.
<u>Return:</u>	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);
```

<u>Parameters:</u>	None.
<u>Return:</u>	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.



### 7.3.1 Supported Types of LF Tags (125 kHz - 134.2 kHz)

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

### 7.3.2 Supported Types of HF Tags (13.56 MHz)

Definition	Frequency	Name	Status
HFTAG_MIFARE	HF	ISO14443A / MIFARE	Supported
HFTAG_ISO14443B	HF	ISO14443B	Supported
HFTAG_ISO15693	HF	ISO15693 / Tag-it	Supported
HFTAG_LEGIC	HF	LEGIC	Supported by TWN4 LEGIC NFC
HFTAG_HIDICLASS	HF	HID iCLASS	Supported, read of UID, read of PAC with option I
HFTAG_FELICA	HF	FeliCa	Supported, read of UID only
HFTAG_SRX	HF	SRC	Supported
HFTAG_NFCP2P	HF	NFC Peer-to-Peer	Supported

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	0-63
HITAG S 2048	2048	256	0-63
HITAG S 256	256	32	0-7

### 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.

<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .
----------------	---

### 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:

<code>int BlockAddress</code>	Specifies the first page address of the block to be written.
<code>byte *Page</code>	Pointer to an array of 4 to 16 bytes which are written to the transponder.
<code>int *BytesWritten</code>	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.

<u>Return:</u>	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 EM4x50-Specific Transponder Operations

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

### 10.1 Functions

Perform a login operation to the transponder.

#### 10.1.1 EM4150\_Login

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

Parameters:

const byte \*Password    Pointer to an array of 4 bytes which contains the password.

Return:                      If the operation was successful, the return value is true, otherwise it is false.

#### 10.1.2 EM4150\_ReadWord

Read one word (4 bytes) from the transponder.

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

Parameters:

int Address                      Specifies the address of the page to be read. The valid address range is from 0 to 33.

byte \*Word                      Pointer to an array of 4 bytes which contains data read from the transponder.

Return:                      If the operation was successful, the return value is true, otherwise it is false.

### 10.1.3 EM4150\_WriteWord

Write one word (4 bytes) to the transponder.

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

Parameters:

int Address	Specifies the address of the page to be written.
const byte *WordData	Pointer to an array of 4 bytes which contains data to be written to the transponder.

Return: If the operation was successful, the return value is true, otherwise it is false.

### 10.1.4 EM4150\_WritePassword

Change the password stored on a transponder.

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

Parameters:

const byte *ActualPassword	Pointer to an array of 4 bytes which specifies the current password of the transponder.
const byte *NewPassword	Pointer to an array of 4 bytes which specifies the password to be written to the transponder.

Return: If the operation was successful, the return value is true, otherwise it is false.

## 11 AT55xx-Specific Transponder Operations

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

### 11.1 Control Functions

#### 11.1.1 AT55\_Begin

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

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

Parameters:                      None.

Return:                              None.

Background:

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

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

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

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

### 11.2 Read Data

Requirements:

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

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

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

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

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

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

### 11.2.1 AT55\_ReadBlock

Read one block (4 bytes) from the transponder.

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

#### Parameters:

int Address	Specifies the address of the page to be read.
byte *Data	Pointer to an array of 4 bytes which contains data read from the transponder.

#### Return:

If the operation was successful, the return value is true, otherwise it is false.

### 11.2.2 AT55\_ReadBlockProtected

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

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

Parameters:

int Address                      Specifies the address of the page to be read.

byte \*Data                      Pointer to an array of 4 bytes which contains data read from the transponder.

const byte \*Password      Pointer to an array of 4 bytes which contains the password.

Return:                      If the operation was successful, the return value is true, otherwise it is false.

## 11.3 Write Data

### 11.3.1 AT55\_WriteBlock

Write one block (4 bytes) to the transponder.

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

Parameters:

int Address                      Specifies the address of the page to be written.

const byte \*Data              Pointer to an array of 4 bytes which contains data to be written to the transponder.

Return:                      If the operation was successful, the return value is true, otherwise it is false.

### 11.3.2 AT55\_WriteBlockProtected

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

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

Parameters:

int Address                      Specifies the address of the page to be written.

const byte \*Data              Pointer to an array of 4 bytes which contains data to be written to the transponder.

const byte \*Password      Pointer to an array of 4 bytes which contains the password.

Return:                      If the operation was successful, the return value is true, otherwise it is false.

### 11.3.3 AT55\_WriteBlockAndLock

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

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

Parameters:

int Address	Specifies the address of the page to be written.
const byte *Data	Pointer to an array of 4 bytes which contains data to be written to the transponder.

Return: If the operation was successful, the return value is true, otherwise it is false.

### 11.3.4 AT55\_WriteBlockProtectedAndLock

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

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

Parameters:

int Address	Specifies the address of the page to be written.
const byte *Data	Pointer to an array of 4 bytes which contains data to be written to the transponder.
const byte *Password	Pointer to an array of 4 bytes which contains the password.

Return: If the operation was successful, the return value is true, otherwise it is false.

## 12 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.

### 12.1 Search Function

#### 12.1.1 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:

<code>int *IDBitCount</code>	A pointer to an integer, which receives the number of actually read bits(!). Due to the nature of the functions <code>TILF_ChargeOnlyRead</code> and <code>TILF_MUGeneralReadPage</code> , the number of received bits is either 32 or 64.
<code>byte *ID</code>	A pointer to an array of bytes, which receives the read ID. Due to the nature of the functions <code>TILF_ChargeOnlyRead</code> and <code>TILF_MUGeneralReadPage</code> , the number of received bytes is either 4 or 8.
<code>int MaxIDBytes</code>	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`.

## 12.2 Single-Page Read/Write Function

### 12.2.1 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.
----------------	---

<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.
----------------	---

### 12.2.2 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.
----------------	--

<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.
----------------	---

### 12.2.3 TILF\_SPProgramPage

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

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



Parameters:

<code>const byte *WriteData</code>	A pointer to an array of 8 bytes, which will be written to the transponder.
<code>byte *ReadData</code>	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.

### 12.2.4 TILF\_SPProgramPageLo

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

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

Parameters:

<code>const byte *WriteData</code>	A pointer to an array of 10 bytes, which will be written to the transponder.
<code>byte *ReadData</code>	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.

## 12.3 Multi-Page Read/Write Function

### 12.3.1 TILF\_MPGeneralReadPage

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

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

Parameters:

<code>int Address</code>	The page address, where data will be read from.
<code>byte *ReadData</code>	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.

### 12.3.2 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.

### 12.3.3 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.

### 12.3.4 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:

<code>int Address</code>	The page address, where data will be programmed to.
<code>const byte *SelectiveAddress</code>	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
<code>const byte *WriteData</code>	A pointer to an array of 8 bytes, which will be programmed.
<code>byte *ReadData</code>	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.

**12.3.5 TILF\_MPLockPage**

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

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

Parameters:

<code>int Address</code>	The page address, which will be locked.
<code>byte *ReadData</code>	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.

**12.3.6 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:

<code>int Address</code>	The page address, which will be locked.
<code>const byte *SelectiveAddress</code>	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
<code>byte *ReadData</code>	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.

### 12.3.7 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.

### 12.3.8 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.

### 12.3.9 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:

<code>int Address</code>	The page address, where data will be programmed to.
<code>const byte *WriteData</code>	A pointer to an array of 10 bytes, which will be programmed.
<code>byte *ReadData</code>	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.

**12.3.10 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:

<code>int Address</code>	The page address, where data will be programmed to.
<code>const byte *SelectiveAddress</code>	Pointer to an array of 3 bytes (24 bits) which provides the selective address.
<code>const byte *WriteData</code>	A pointer to an array of 10 bytes, which will be programmed.
<code>byte *ReadData</code>	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.

**12.3.11 TILF\_MPLockPageLo**

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

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

Parameters:

<code>int Address</code>	The page address, which will be locked.
<code>byte *ReadData</code>	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.

### 12.3.12 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.

## 12.4 Multi-Usage Read/Write Function

### 12.4.1 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.

### 12.4.2 TILF\_MUSelectiveReadPage

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

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

Parameters:

<code>int Address</code>	The page address, where data will be read from.
<code>int SelectiveAddress</code>	A value which specifies the 8-bit selective address.
<code>byte *ReadData</code>	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.

**12.4.3 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:

<code>int Address</code>	The page address, where data will be read from.
<code>const byte *SpecialAddress1</code>	Pointer to an array of 5 bytes (40 bits) which provides the special address 1.
<code>const byte *SpecialAddress2</code>	Pointer to an array of 3 bytes (24 bits) which provides the special address 2.
<code>byte *ReadData</code>	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.

**12.4.4 TILF\_MUProgramPage**

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

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

Parameters:

<code>int Address</code>	The page address, where data will be programmed to.
<code>const byte *WriteData</code>	A pointer to an array of 5 bytes, which will be programmed.
<code>byte *ReadData</code>	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.

### 12.4.5 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.
<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.

### 12.4.6 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.
<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.

### 12.4.7 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.
<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.

### 12.4.8 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.
<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.

### 12.4.9 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.
<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false.

## 13 ISO14443 Transponder Operations

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

### 13.1 ISO14443A

#### 13.1.1 Get ATQA

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

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

Parameters:

`byte* ATQA`

After successful completion of this function, the buffer referred by this parameter holds the ATQA of the transponder. The function returns two bytes of data.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`.

#### 13.1.2 Get SAK

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

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

Parameters:

`byte*` SAK                      After successful completion of this function, the buffer referred by this parameter holds the SAK of the transponder. The function returns one byte of data.

Return:                      If the operation was successful, the return value is `true`, otherwise it is `false`.

### 13.1.3 Get ATS

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

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

Parameters:

`byte*` ATS                      After successful completion of this function, the buffer referred by this parameter holds the ATS which was read from the transponder. Take care for adequate dimensioning.

`int*` ATSByteCnt              After successful completion of this function, this parameter holds the number of bytes, the ATS contains.

`int` MaxATSByteCnt            This parameter holds the array-size of ATS in bytes.

Return:                      If the operation was successful, the return value is `true`, otherwise it is `false`.

## 13.2 ISO14443B

### 13.2.1 Get ATQB

This function delivers the ATQB (Answer To Request TypeB) of the last detected ISO14443B compliant transponder.

Note: This function can only be called on TWN4 MIFARE NFC.

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

Parameters:

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

**13.2.2 Get Answer to ATTRIB**

This function delivers the transponder's answer to the ATTRIB command, which is sent automatically during selection process by the reader.

Note: This function can only be called on TWN4 MIFARE NFC.

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

Parameters:

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

**13.3 Check Presence**

This function can be used to probe if a ISO14443-4 transponder is still in reading range. The internal state of the transponder remains unchanged.

Note: This function can only be called on TWN4 MIFARE NFC.

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

Parameters: None.

Return: If the transponder is still in range, the return value is `true`, otherwise it is `false`.

## 13.4 ISO14443-4 Transparent Data Exchange

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

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

Parameters:

`byte* TXRX` This buffer holds the byte-string that shall be transmitted to the transponder. The response of the transponder is also returned by this parameter. Take care for adequate dimensioning.

`int TXByteCnt` This parameter holds the number of bytes that shall be transmitted to the transponder.

`int* RXByteCnt` After successful completion of this function, this parameter holds the number of bytes that the transponder response contains.

`int MaxRXByteCnt` This parameter holds the array-size of TXRX in bytes.

Return: If the operation was successful, the return value is `true`, otherwise it is `false`.

## 14 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.

### 14.1 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:

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

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 14.1: Well-known keys for MIFARE Classic transponders

## 14.2 Read/Write Data

### 14.2.1 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:

<code>int</code> Block	Specify the address of the block to be read. The valid range of this parameter is between 0 and 255.
<code>byte*</code> 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`.

### 14.2.2 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:

<code>int</code> Block	Specify the address of the block to be written. The valid range of this parameter is between 0 and 255.
<code>const byte*</code> 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`.

## 14.3 Handling of Value Blocks

### 14.3.1 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`.

### 14.3.2 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`.

### 14.3.3 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`.

### 14.3.4 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`.

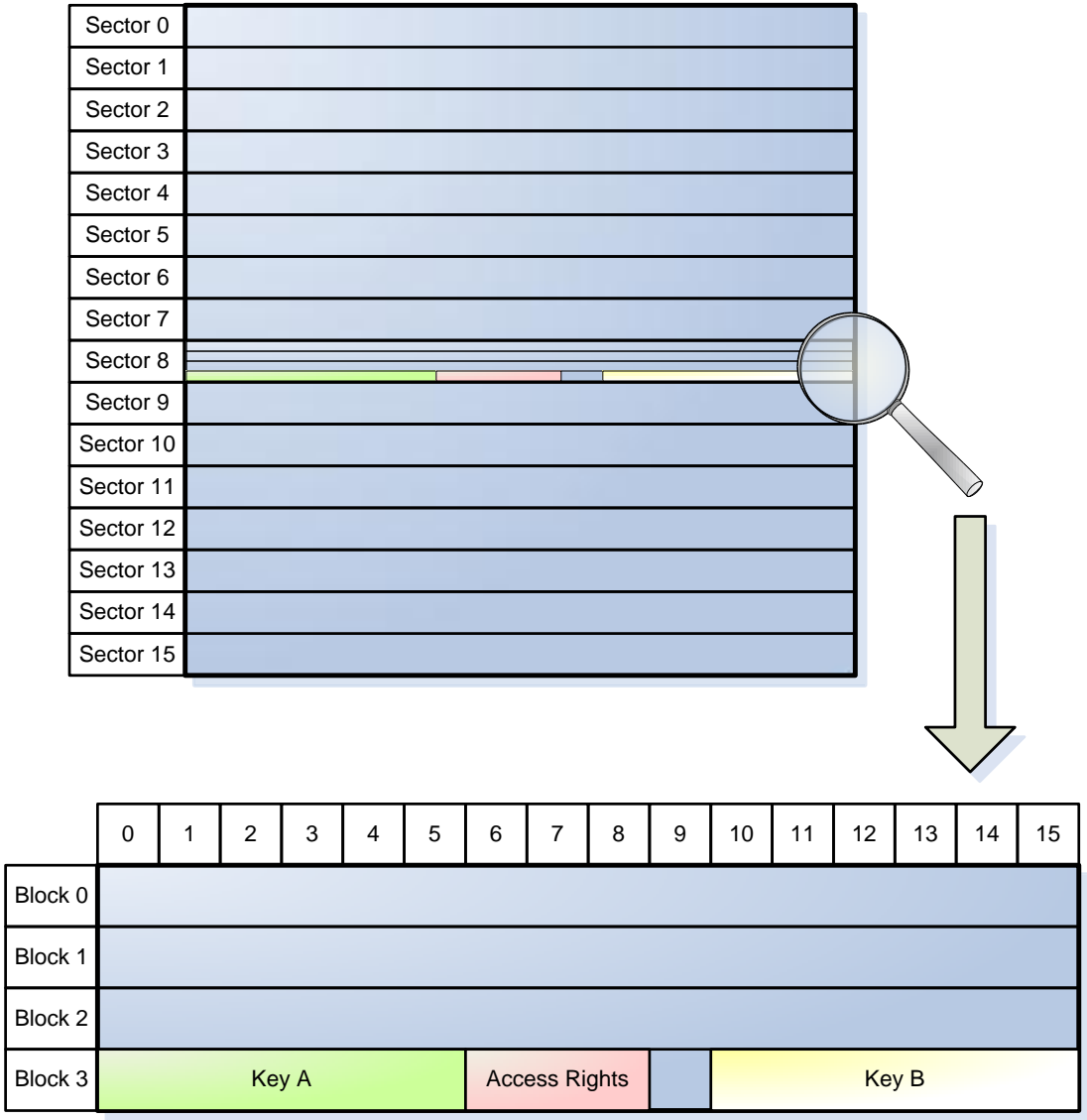


Figure 14.1: Memory layout of a MIFARE Classic 1K transponder

# 15 MIFARE Ultralight/Ultralight C Specific Transponder Operations

## 15.1 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 15.1: Well-known key for MIFARE Ultralight C transponders

## 15.2 Read/Write Data

### 15.2.1 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`.

### 15.2.2 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`.

## 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-

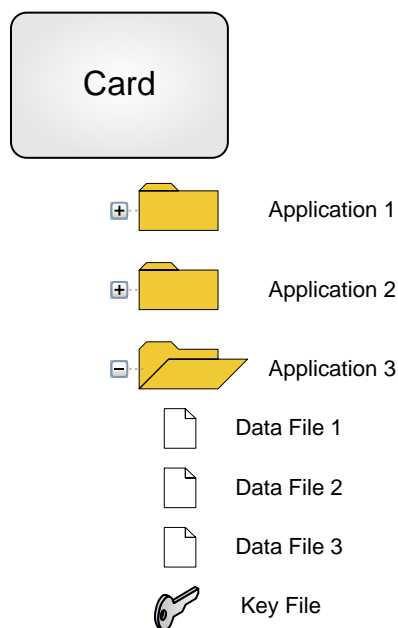


Figure 16.1: DESFire memory layout

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

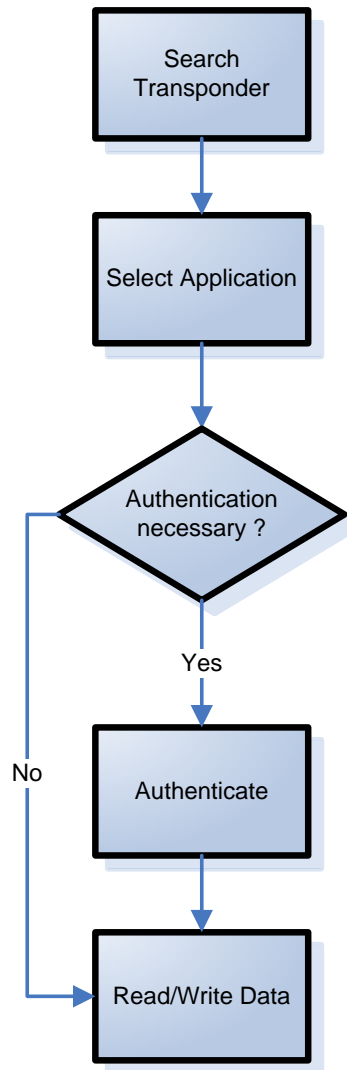


Figure 16.2: Simple way to gain access to the file system

## 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:

<code>int</code> 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.
<code>int</code> 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.
<code>const byte*</code> 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.
<code>int</code> KeyByteCount	Specify the key length of the key. Use one of the predefined constants DESF_KEYLEN_3DES, DESF_KEYLEN_3K3DES or DESF_KEYLEN_AES.
<code>int</code> 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.
<code>int</code> 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.
<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .

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:

<code>int CryptoEnv</code>	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.
<code>int KeyNoTag</code>	Specify the key number that shall be queried.
<code>byte* KeyVer</code>	The key version information is returned as one byte by this parameter.

<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .
----------------	---

#### 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
<code>byte</code> AllowChangeMasterKey	1	If set to 1 the master key is changeable, otherwise it cannot be changed any more.
<code>byte</code> 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.
<code>byte</code> 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.
<code>byte</code> 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.
<code>byte</code> ChangeKeyAccessRights	4	<p>This member holds the access rights for changing keys. On transponder level this member is set to 0.</p> <p>0x0: Authentication with the master key is necessary to change any key.</p> <p>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.</p> <p>0xE: Authentication with the key to be changed is necessary to change the key.</p> <p>0xF: All keys except the master key are frozen.</p>

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:

<code>int CryptoEnv</code>	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.
<code>int KeyNoTag</code>	Specify the key number that shall be changed.
<code>const byte* OldKey</code>	Specify the old key.
<code>int OldKeyByteCount</code>	Specify the length of the old key in bytes.
<code>const byte* NewKey</code>	Specify the new key.
<code>int NewKeyByteCount</code>	Specify the length of the new key in bytes.
<code>byte KeyVersion</code>	Specify the key version of the new key.
<code>const TDESFireMasterKeySettings* MasterKeySettings</code>	This structure holds the current master key settings. See chapter <i>Get Key Settings</i> 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:

<code>int</code> 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.
<code>int</code> 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.
<code>const</code> TDESFireMasterKeySettings* MasterKeySettings	This structure holds the master key settings of the new application. See chapter <i>Get Key Settings</i> 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:

<code>int</code> 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.
<code>int*</code> AIDs	After successful completion of this function, this parameter holds a list of the retrieved application IDs.
<code>int*</code> NumberOfAIDs	This parameter holds the number of retrieved application IDs.
<code>int</code> 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	112	This member holds manufacturing specific information.

Table 16.3: Definition of TDESFireVersion

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

Table 16.4: Definition of TDESFireTagInfo

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

<code>int</code> 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.
<code>const byte*</code> Key	This parameter specifies the new default key.
<code>int</code> 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.
<code>byte</code> 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:

<code>int</code> 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.
<code>const byte*</code> ATS	This parameter specifies the new ATS.
<code>int</code> 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
<code>byte</code> <code>FileType</code>	8	This member defines the file type. Possible values are: <code>DESF_FILETYPE_STDDATAFILE</code> , <code>DESF_FILETYPE_BACKUPDATAFILE</code> , <code>DESF_FILETYPE_VALUEFILE</code> , <code>DESF_FILETYPE_LINEARRECORDFILE</code> , <code>DESF_FILETYPE_CYCLICRECORDFILE</code> .
<code>byte</code> <code>CommSet</code>	8	This member defines the communication settings between reader and transponder when the file is accessed. Possible values are: <code>DESF_COMMSET_PLAIN</code> , <code>DESF_COMMSET_PLAIN_MACED</code> , <code>DESF_COMMSET_FULLY_ENC</code> .
<code>uint16_t</code> <code>AccessRights</code>	16	This member holds the access rights.
<code>union</code> <code>TDESFireSpecificFileInfo</code> <code>SpecificFileInfo</code>	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
<code>struct</code> TDESFireDataFileSettings DataFileSettings	32	Definition of data file settings.
<code>struct</code> TDESFireValueFileSettings ValueFileSettings	128	Definition of value file settings.
<code>struct</code> TDESFireRecordFileSettings RecordFileSettings	96	Definition of record file settings.

Table 16.8: Definition of `union` TDESFireSpecificFileInfo

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

Table 16.9: Definition of `struct` TDESFireDataFileSettings

Members	Length (Bits)	Description
<code>uint32_t</code> LowerLimit	32	Definition of the lower limit which must not be passed by a debit operation.
<code>uint32_t</code> UpperLimit	32	Definition of the upper limit which must not be passed by a credit operation.
<code>uint32_t</code> LimitedCreditValue	32	Definition of the initial value of the file at file creation.
<code>bool</code> 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*`      This member holds the file settings. See description of  
`FileSettings`      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:

<code>int CryptoEnv</code>	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.
<code>int FileNo</code>	Specify the ID of the file which shall be deleted. If the ID doesn't exist within the application, this results in an error.

<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .
----------------	---

### 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:

<code>int</code> 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.
<code>byte*</code> FileIDList	After successful completion of this function, this parameter holds a list of the retrieved file IDs.
<code>int*</code> FileIDCount	This parameter holds the number of retrieved file IDs.
<code>int</code> 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:

<code>int</code> 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.
<code>int</code> FileNo	Specify the file ID which shall be queried.
<code>TDESFireFileSettings*</code> <code>FileSettings</code>	This member holds the returned file settings. See description of <code>TDESFireFileSettings</code> for details.
<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .

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:

<code>int CryptoEnv</code>	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.
<code>int FileNo</code>	Specify the file ID whose settings shall be changed.
<code>int NewCommSet</code>	Specify the new communication settings. Possible values are: DESF_COMMSET_PLAIN, DESF_COMMSET_PLAIN_MACED, DESF_COMMSET_FULLY_ENC.
<code>int OldAccessRights</code>	Specify the current Access Rights of the file.
<code>int NewAccessRights</code>	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:

<code>int</code> 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.
<code>int</code> FileNo	Specify the ID of the file that shall be read.
<code>byte*</code> 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.
<code>int</code> 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.
<code>int</code> 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.
<code>int</code> 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:

<code>int CryptoEnv</code>	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.
<code>int FileNo</code>	Specify the ID of the file that shall be written.
<code>const byte* Data</code>	The buffer referred by this parameter holds the data which is written to the file.
<code>int Offset</code>	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.
<code>int Length</code>	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.
<code>int CommSet</code>	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:

<code>int</code> 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.
<code>int</code> FileNo	Specify the ID of the Value File whose value shall be queried.
<code>int*</code> Value	After successful completion of this function, this parameter holds the value which was read from the file.
<code>int</code> 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:

<code>int</code> 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.
<code>int</code> FileNo	Specify the ID of the Value File that shall be debited.
<code>const int</code> Value	The value stored in the value file will be decreased by this parameter.
<code>int</code> 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:

<code>int</code> 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.
<code>int</code> FileNo	Specify the ID of the Value File that shall be credited.
<code>const int</code> Value	The value stored in the value file will be increased by this parameter.
<code>int</code> 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:

<code>int</code> 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.
<code>int</code> FileNo	Specify the ID of the Value File that shall be credited.
<code>const int</code> 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.
<code>int</code> 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:

<code>int</code> 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`.

## 17 ISO15693 Specific Transponder Operations

### 17.1 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:

<code>byte</code> Flags	Specify the ISO15693 flags. Note: The flags regarding RF-communication are set automatically, so by default one may assign 0x00 to this parameter.
<code>byte</code> Command	Command code.
<code>byte*</code> 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.
<code>int*</code> Length	This parameter works as Input/Output-variable. It holds the payload-length of Data in the directions <i>Reader→Tag</i> and <i>Tag→Reader</i> .
<code>int</code> BufferSize	This parameter holds the array-size of Data in bytes.
<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .

## 17.2 Gather Tag Specific Information

### 17.2.1 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\*    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.

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

Table 17.1: Definition of TISO15693\_SystemInfo

### 17.2.2 Get Tag Type

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

#### 17.2.2.1 Get Tag Type From UID

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

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

##### Parameters:

`byte* UID`

This parameter holds the UID. Watch for the correct byte order; UID[0] shall have the value 0xE0

##### Return:

The return-value is the determined tag-type which is represented by one of the constants in the table below.

Definition	Value	Manufacturer	Tag Type
ISO15693_TAGTYPE_ICODESL2	0x00	NXP	ICode SL2
ISO15693_TAGTYPE_ICODESL2S	0x01		ICode SL2S
ISO15693_TAGTYPE_UNKNOWNNXP	0x0F		Unknown
ISO15693_TAGTYPE_TAGITHFIPLUSINLAY	0x10	TI	Tag-It HFI Plus Inlay
ISO15693_TAGTYPE_TAGITHFIPLUSCHIP	0x11		Tag-It HFI Plus Chip
ISO15693_TAGTYPE_TAGITHFISTD	0x12		Tag-It HFI Standard
ISO15693_TAGTYPE_TAGITHFIPRO	0x13		Tag-It HFI Pro
ISO15693_TAGTYPE_UNKNOWNNTI	0x1F		Unknown
ISO15693_TAGTYPE_UNKNOWNST	0x4F	ST	Unknown
ISO15693_TAGTYPE_SRF55V02P	0x50	Infineon	SRF55V02P
ISO15693_TAGTYPE_SRF55V10P	0x51		SRF55V10P
ISO15693_TAGTYPE_SRF55V02S	0x52		SRF55V02S
ISO15693_TAGTYPE_SRF55V10S	0x53		SRF55V10S
ISO15693_TAGTYPE_UNKNOWNINFINEON	0x5F		Unknown
ISO15693_TAGTYPE_UNKNOWN	0xFF	Unknown	Unknown ISO15693

Table 17.2: Retrievable tag types from UID



### 17.2.2.2 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\*    Pointer to the structure which holds the System Information.  
SystemInfo

Return:

The return-value is the determined tag-type which is represented by one of the constants in the table below.

Definition	Value	Manufacturer	Tag Type
ISO15693_TAGTYPE_ICODESL2	0x00	NXP	ICode SL2
ISO15693_TAGTYPE_ICODESL2S	0x01		ICode SL2S
ISO15693_TAGTYPE_UNKNOWNNXP	0x0F		Unknown
ISO15693_TAGTYPE_TAGITHFIPLUSINLAY	0x10	TI	Tag-It HFI Plus Inlay
ISO15693_TAGTYPE_TAGITHFIPLUSCHIP	0x11		Tag-It HFI Plus Chip
ISO15693_TAGTYPE_TAGITHFISTD	0x12		Tag-It HFI Standard
ISO15693_TAGTYPE_TAGITHFIPRO	0x13		Tag-It HFI Pro
ISO15693_TAGTYPE_UNKNOWNNTI	0x1F		Unknown
ISO15693_TAGTYPE_MB89R118	0x20	Fuji	MB89R118
ISO15693_TAGTYPE_MB89R119	0x21		MB89R119
ISO15693_TAGTYPE_MB89R112	0x22		MB89R112
ISO15693_TAGTYPE_UNKNOWNFUJI	0x2F		Unknown
ISO15693_TAGTYPE_24LR16	0x30	ST	24LR16
ISO15693_TAGTYPE_24LR64	0x31		24LR64
ISO15693_TAGTYPE_LRI1K	0x40		LRI1K
ISO15693_TAGTYPE_LRI2K	0x41		LRI2K
ISO15693_TAGTYPE_LRIS2K	0x42		LRIS2K
ISO15693_TAGTYPE_LRIS64K	0x43		LRIS64K
ISO15693_TAGTYPE_UNKNOWNST	0x4F		Unknown
ISO15693_TAGTYPE_SRF55V02P	0x50	Infineon	SRF55V02P
ISO15693_TAGTYPE_SRF55V10P	0x51		SRF55V10P
ISO15693_TAGTYPE_SRF55V02S	0x52		SRF55V02S
ISO15693_TAGTYPE_SRF55V10S	0x53		SRF55V10S
ISO15693_TAGTYPE_UNKNOWNINFINEON	0x5F		Unknown
ISO15693_TAGTYPE_UNKNOWN	0xFF	Unknown	Unknown ISO15693

Table 17.3: Retrievable tag types from System Information

## 17.3 Read/Write Data

### 17.3.1 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:

<code>int</code> BlockNumber	This parameter holds the number of the block to be read.
<code>byte*</code> 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.
<code>int*</code> Length	This parameter holds the length of data which was read from the tag in bytes.
<code>int</code> 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`.

### 17.3.2 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:

<code>int</code> BlockNumber	This parameter holds the number of the block to be written.
<code>const byte*</code> BlockData	This parameter holds the data which shall be written to the tag.
<code>int</code> 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`.

## 18 LEGIC-Specific Functions

This chapter describes functions for accessing LEGIC functionality.

Notes:

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

### 18.1 Direct Access of LEGIC Chip

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

Note:

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

#### 18.1.1 SM4X00\_GenericRaw

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

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

Parameters:

<code>const byte *TXData</code>	Pointer to an array of bytes, which contains the command to be sent to SM4X00.
<code>int TXDataLength</code>	Number of bytes to be sent to SM4X00.
<code>byte *RXData</code>	Pointer to an array of bytes, which receives response from SM4X00
<code>int *RXDataLength</code>	Pointer to an integer, which receives the actually read number of bytes.
<code>int MaxRXDataLength</code>	A value, which specifies the maximum number of bytes, which can be received byte RXData, thus the buffer size.
<code>int Timeout</code>	Maximum time, the function should wait for a response from SM4X00. This value is specified in milliseconds.

Return: If the operation was successful, the return value is `true`, otherwise it is `false`.

**18.1.2 SM4X00\_Generic**

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

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

Parameters:

<code>const byte *TXData</code>	Pointer to an array of bytes, which contains the command to be sent to SM4X00. The command has to be specified W/O leading length byte and W/O closing CRC value.
<code>int TXDataLength</code>	Number of bytes contained in TXData.
<code>byte *RXData</code>	Pointer to an array of bytes, which receives response from SM4X00. Received data is W/O length byte and W/O CRC value.
<code>int *RXDataLength</code>	Pointer to an integer, which receives length of the actually received payload.
<code>int MaxRXDataLength</code>	A value, which specifies the maximum number of bytes, which can be received by RXData, thus the buffer size.
<code>int Timeout</code>	Maximum time, the function should wait for a response from SM4X00. This value is specified in milliseconds.

Return:

If the operation was successful, the return value is true, otherwise it is false.

**18.1.3 SM4X00\_StartBootloader**

Start boot loader of SM4X00.

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

Parameters:

`byte *TLV`  
`int *TLVLength`  
`int MaxTLVLength`

Return:

If the operation was successful, the return value is true, otherwise it is false.

**18.1.4 SM4X00\_EraseFlash**

Erase flash of SM4X00.

```
bool SM4X00_EraseFlash(void)
```

Parameters:

None.

Return:

If the operation was successful, the return value is true, otherwise it is false.

### 18.1.5 SM4X00\_ProgramBlock

Program one block of data into the flash of SM4X00.

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

Parameters:

byte *Data	Pointer to an array of bytes.
bool *Done	Pointer to a boolean variable, which receives the status, if the last block was flashed.

Return:

If the operation was successful, the return value is true, otherwise it is false.



## 19 iCLASS Specific Transponder Operations

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

- The reader must be the TWN4 MIFARE NFC version
- An iCLASS SIO card must be inserted into one of the SAM slots
- The I-Option must be activated

### 19.1 Read PAC Bits

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

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

#### Parameters:

<code>byte*</code> PACBits	After successful completion of this function, the buffer referred by this parameter holds the PAC bits read from the transponder. Take care for adequate dimensioning.
<code>int*</code> PACBitCnt	After successful completion of this function, this parameter holds the number of bits, the PAC contains.
<code>int</code> MaxPACBytes	This parameter holds the maximum number of bytes which the buffer PACBits can hold.

Return: If the operation was successful, the return value is `true`, otherwise it is `false`.

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

## 19.2 Example

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

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

byte PACBits[8];
int PACBitCnt;

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

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

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

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

## 20 Simple NDEF Exchange Protocol (SNEP)

This chapter handles transmission of NDEF (NFC Data Exchange Format) messages between a TWN4 reader and a NFC enabled device using the Simple NDEF Exchange Protocol. For message exchange, a NFC Peer-to-Peer connection must have been established. The SNEP service provides both a logical In-Box and a logical Out-Box for receiving and transmitting messages. Each message box works as FIFO, which enables reader and host-software to exchange even large messages as a stream of data. This functionality is also useful to reduce outbound buffering on host side. Each message box can manage only one message at the same time, so message-queuing is currently not supported. Large messages that do not fit into the internal FIFO must be transmitted fragmented, so the sending side must break up the message into smaller parts that fit into the FIFO, the receiving side must reassemble the parts as a consequence. When dealing with large messages, it might become necessary to read data from the FIFO fast enough during a ongoing transmission in order to prevent any tailbacks.

Note: This functionality is only available on TWN4 MIFARE NFC.

### 20.1 Initialize SNEP Service

Use this function for initialization and starting of the built-in SNEP service. The function should be called at last once before issuing `SearchTag()` with `TagType HFTAG_NFCP2P` enabled. Depending on the implementation of the counterpart NFC device, there might be a delay until the SNEP service is activated on both communication peers. This time usually ranges around 100 ms up to 500 ms.

```
bool SNEP_Init(void);
```

<u>Parameters:</u>	None.
<u>Return:</u>	If the SNEP service was successfully started, the return value is <code>true</code> , otherwise it is <code>false</code> .

## 20.2 Get Connection State

Use this function to query the current connection state of the SNEP service. This can be used for checking e.g. any loss of the physical NFC Peer-to-Peer connection.

```
int SNEP_GetConnectionState(void);
```

<u>Parameters:</u>	None.
<u>Return:</u>	<p>SNEP_STATE_DEINIT: The SNEP service has not been started.</p> <p>SNEP_STATE_SLEEP: The SNEP service has been started, but there is no active connection.</p> <p>SNEP_STATE_IDLE: The SNEP service is running, but there is currently no active exchange of messages.</p> <p>SNEP_STATE_CONNCLIENT: The SNEP service is running in client mode.</p> <p>SNEP_STATE_CONNSERVER: The SNEP service is running in server mode.</p>

## 20.3 Query Message FIFO

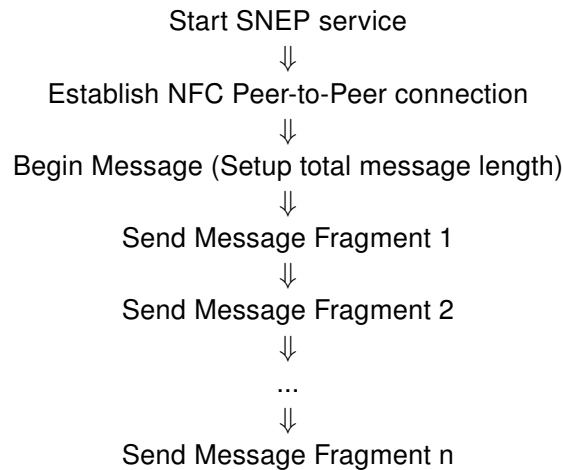
Use this function to get information of the respective message FIFO.

```
int SNEP_GetFragmentByteCount(int Direction);
```

<u>Parameters:</u>	
<code>int</code> Direction	Specify the message box to be queried by this parameter. Valid values are DIR_OUT (Out-Box) or DIR_IN (In-Box), use one of these predefined constants.
<u>Return:</u>	If the In-Box is queried, the return value is the current number of bytes which are available for reading from host side. If the Out-Box is queried, the return value is the number of bytes that can be written to the FIFO.

## 20.4 Transmit NDEF Message

This section handles transmission of NDEF messages. A typical communication flow for transmitting a NDEF message looks like this:



### 20.4.1 Begin Message

Use this function to setup the total message length. A message can reach up to 4 GBytes.

```
bool SNEP_BeginMessage(uint32_t MsgByteCnt);
```

#### Parameters:

uint32\_t MsgByteCnt      Specify the total message length by this parameter.

#### Return:

If the operation was successful, the return value is `true`. If a previously set up message has not been transmitted completely, the return value is `false`.

### 20.4.2 Send Message Fragment

Use this function to store a fragment of a message in the Out-Box FIFO. The message must be transmitted completely in order to make the FIFO available for new outgoing messages.

```
bool SNEP_SendMessageFragment(const byte* MsgFrag, int FragByteCnt);
```

Parameters:

`const byte*` MsgFrag      Specify the buffer that holds the message fragment by this parameter.

`int` FragByteCnt      This parameter holds the length the message fragment.

Return:      If the operation was successful, the return value is `true`, otherwise it is `false`.

### 20.4.3 Example

The following example demonstrates transmission of a NDEF message from a TWN4 MI-FARE NFC reader to another NFC device running SNEP:

```
const byte NDEF_Message[] = { /* Your NDEF message */ };

void TransmitNDEFMessage(void)
{
    // Wait for SNEP service is running
    unsigned long SNEPConnectionStartTime = GetSysTicks();
    // SNEP service must be at least in IDLE state
    while (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
    {
        if (GetSysTicks() - SNEPConnectionStartTime > 500)
            return;
    }
    // Transmit NDEF message as long as a NFC connection is established
    int FragmentOffset = 0;
    int NDEF_MessageByteCnt = sizeof(NDEF_Message);
    while (true)
    {
        if (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
            return;
        // Get available buffer size from operating system for message fragmenting
        FragmentSize = SNEP_GetFragmentByteCount(DIR_OUT);
        if (FragmentSize > 0)
        {
            // Is this the first fragment?
            if (FragmentOffset == 0)
            {
                // Yes, Setup message
                if (!SNEP_BeginMessage(NDEF_MessageByteCnt))
                    return;
            }
            // Calculate fragment size
            if (NDEF_MessageByteCnt - FragmentOffset <= FragmentSize)
                FragmentSize = NDEF_MessageByteCnt - FragmentOffset;
            // Send a fragment of the message
            if (!SNEP_SendMessageFragment(&NDEF_Message[FragmentOffset], FragmentSize))
                return;
            FragmentOffset += FragmentSize;
        }
        // Was the message completely transmitted?
        if (FragmentOffset == NDEF_MessageByteCnt)
            return;
    }
}
```

```
#define MAXIDBYTES 10

byte ID[MAXIDBYTES];
int IDBitCnt;
int TagType;

int main(void)
{
    // Enable NFC Peer-to-Peer mode
    SetTagTypes(0, TAGMASK(HFTAG_NFCP2P));

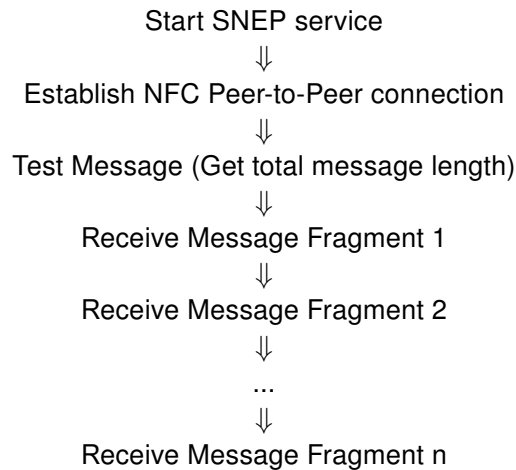
    // Start SNEP service
    SNEP_Init();

    while (true)
    {
        // Search a transponder
        if (SearchTag(&TagType, &IDBitCnt, ID, sizeof(ID)))
        {
            if (TagType == HFTAG_NFCP2P)
            {
                // Transmit NDEF message
                TransmitNDEFMessage();
            }
        }
    }
}
```



## 20.5 Receive NDEF Message

This section handles reception of NDEF messages. A typical communication flow for receiving a NDEF message looks like this:



### 20.5.1 Test Message

Use this function to test if there is a new message available in the In-Box. The function returns the total length of the message. A message can reach up to 4 GBytes.

```
bool SNEP_TestMessage(uint32_t* MsgByteCnt);
```

Parameters:

uint32\_t\* MsgByteCnt      The total message length is returned by this parameter.

Return:                      If a message is available, the return value is true, otherwise it is false.

### 20.5.2 Receive Message Fragment

Use this function to receive a fragment of a message stored in the In-Box FIFO. A message must be read completely from the FIFO in order to make it available for new incoming messages.

```
bool SNEP_ReceiveMessageFragment(byte* MsgFrag, int FragByteCnt);
```

Parameters:

`byte*` MsgFrag

Specify the buffer that holds the message fragment by this parameter.

`int` FragByteCnt

This parameter holds the length the message fragment to be read.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`.

### 20.5.3 Example

The following example demonstrates reception of a NDEF message from another NFC device running SNEP:

```
void ReceiveNDEFMessage(void)
{
    // Wait for SNEP service is running
    unsigned long SNEPConnectionStartTime = GetSysTicks();
    // SNEP service must be at least in IDLE state
    while (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
    {
        if (GetSysTicks() - SNEPConnectionStartTime > 500)
            return;
    }
    // Receive all NDEF messages as long as a NFC connection is established
    while (true)
    {
        uint32_t MessageSize;
        byte Message[4096];
        // Wait for a incoming NDEF message or loss of connection
        while (!SNEP_TestMessage(&MessageSize))
        {
            if (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
                return;
        }
        // A NDEF message was announced. Now read it.
        int FragmentOffset, FragmentSize;
        for (FragmentOffset = 0; FragmentOffset < MessageSize; FragmentOffset += FragmentSize)
        {
            // Wait, till fragment of the message arrives
            do
            {
                if (SNEP_GetConnectionState() < SNEP_STATE_IDLE)
                    return;
                FragmentSize = SNEP_GetFragmentByteCount(DIR_IN);
            }
            while (FragmentSize == 0);
            SNEP_ReceiveMessageFragment(&Message[FragmentOffset], FragmentSize);
        }
        // We read the entire NDEF message
    }
}

#define MAXIDBYTES 10

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

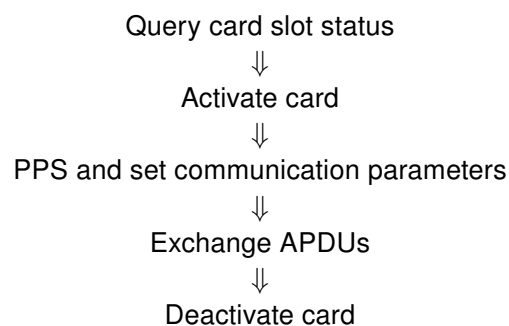
```
int main(void)
{
    // Enable NFC Peer-to-Peer mode
    SetTagTypes(0, TAGMASK(HFTAG_NFCP2P));

    // Start SNEP service
    SNEP_Init();

    while (true)
    {
        // Search a transponder
        if (SearchTag(&TagType, &IDBitCnt, ID, sizeof(ID)))
        {
            if (TagType == HFTAG_NFCP2P)
            {
                // Receive NDEF message
                ReceiveNDEFMessage();
                DoSomething();
            }
        }
    }
}
```

## 21 Contact Card Operations

This chapter handles the usage of ISO7816 compliant Integrated Circuit Cards such as ID-1 or SAM (Secure Access Module) cards. The TWN4 ISO7816 API offers different system functions for covering different imaginable scenarios. A typical communication flow with contact cards looks like this:



### 21.1 Query Card Slot Status

This function shall be used to query information of the physical card slot status, e.g. to find out if a card is inserted or not. The function returns the slot status in CCID compliant style, this means it return information about slot status, error information and clock status. The internal state of the card is not changed. Please note, depending on the used hardware (TWN4 Desktop or TWN4 SmartCard) the amount of retrievable information differs.

```
bool ISO7816_GetSlotStatus(int Channel, TISO7816SlotStatus* SlotStatus);
```

Members	Length (Bits)	Description
TIS07816StatusReg bStatus	8	Slot status register compliant to CCID. See the definition of TIS07816StatusReg for meaning of the different bit fields.
byte bError	8	Error code compliant to CCID.
byte bClockStatus	8	Clock status information compliant to CCID. Possible values are: TIS07816_CLOCKSTATUS_RUNNING, TIS07816_CLOCKSTATUS_CLKSTPL, TIS07816_CLOCKSTATUS_CLKSTPH, TIS07816_CLOCKSTATUS_CLKSTPU.

Table 21.1: Definition of TIS07816SlotStatus

Members	Length (Bits)	Description
byte bmICCStatus	2	Physical status of the card slot. Possible values are: TIS07816_ICCPRESENTANDACTIVE, TIS07816_ICCPRESENTANDINACTIVE and TIS07816_NOICCPRESENT.
byte bmRFU	4	These bits are reserved for future use.
byte bmCommandStatus	2	Command status information compliant to CCID.

Table 21.2: Definition of TIS07816StatusReg

Parameters:`int` Channel

Specify a communication channel by this parameter. Valid values are CHANNEL\_SAM1 through CHANNEL\_SAM4 or CHANNEL\_SC1, use one of these predefined constants.

TIS07816SlotStatus\*  
SlotStatus

The card slot status is returned by this parameter. See the definition of TIS07816SlotStatus for meaning of each member.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`.

## 21.2 Card Activation

This function shall be used to activate and initialize communication with the card inserted in one of the slots connected to the TWN4 reader. All communication parameters are reset to default. Depending on the hardware platform, the reader shows different behaviour regarding reset-handling of the card: On TWN4 Desktop, calling this function always leads to a warm reset, on TWN4 SmartCard, the first call performs a cold reset and any subsequent function call leads to a warm reset until the card is deactivated. The result of the entire operation is the receipt of the Answer To Reset (ATR) from the card. Based on the content of the ATR, the user may decide how to further proceed with the card. Note that selection of voltage level is only available for TWN4 SmartCard.

```
bool ISO7816_IccPowerOn
(
    int Channel,
    byte* ATR,
    int* ATRByteCnt,
    int MaxATRByteCnt,
    byte bPowerSelect,
    TISO7816StatusReg* bStatus,
    byte* bError
);
```

Parameters:

<code>int</code> Channel	Specify a communication channel by this parameter. Valid values are CHANNEL_SAM1 through CHANNEL_SAM4 or CHANNEL_SC1, use one of these predefined constants.
<code>byte*</code> ATR	After successful completion of this function, the buffer referred by this parameter holds the ATR which was read from the card. Take care for adequate dimensioning.
<code>int*</code> ATRByteCnt	After successful completion of this function, this parameter holds the number of bytes, the ATR contains.
<code>int</code> MaxATRByteCnt	This parameter holds the array-size of ATR in bytes.
<code>byte</code> bPowerSelect	Specify the operating voltage level which shall be used for the card. Valid values are ISO7816_POWERSELECT_AUTO, ISO7816_POWERSELECT_5V, ISO7816_POWERSELECT_3V, or ISO7816_POWERSELECT_1V8, use one of these predefined constants.
TISO7816StatusReg* bStatus	The CCID compliant slot status register is returned by this parameter. See the definition of TISO7816StatusReg for meaning of the different bit fields.
<code>byte*</code> bError	The CCID compliant error code is returned by this parameter.
<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .

## 21.3 Card Deactivation

This function shall be used to deactivate and power off the card. When this function was called on TWN4 SmartCard reader, a subsequent call of `IccPowerOn()` leads to a cold reset of the card.

```
bool ISO7816_IccPowerOff(int Channel, TISO7816SlotStatus* SlotStatus);
```



Parameters:

<code>int</code> Channel	Specify a communication channel by this parameter. Valid values are CHANNEL_SAM1 through CHANNEL_SAM4 or CHANNEL_SC1, use one of these predefined constants.
<code>TISO7816SlotStatus*</code> <code>SlotStatus</code>	The card slot status is returned by this parameter. See the definition of <code>TISO7816SlotStatus</code> for meaning of each member.

Return: If the operation was successful, the return value is `true`, otherwise it is `false`.

## 21.4 Set Communication Settings

This function shall be used to assign new communication settings to the respective card slot. After calling this function, the communication parameters which have been negotiated with the card during Protocol And Parameter Selection (PPS) become valid. For issuing a PPS, please refer to the function `ISO7816_Transceive`. Specific communication parameters must be obtained from the ATR, for detailed information refer to standard ISO7816-3.

```
bool ISO7816_SetCommSettings
(
    int Channel,
    const TISO7816CommSettings* CommSettings
);
```

Parameters:

<code>int</code> Channel	Specify a communication channel by this parameter. Valid values are CHANNEL_SAM1 through CHANNEL_SAM4 or CHANNEL_SC1, use one of these predefined constants.
<code>const</code> <code>TISO7816CommSettings*</code> <code>CommSettings</code>	The new communication settings are passed by this parameter. See the definition of <code>TISO7816CommSettings</code> for meaning of each member.

Return: If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
<code>byte</code> Protocol	8	Specify the protocol to be used. Possible values are: <code>ISO7816_PROTOCOL_T0</code> and <code>ISO7816_PROTOCOL_T0</code> .
<code>byte</code> Freq	8	Specify the clock frequency which shall be applied to the card. Chose one of the predefined constants <code>ISO7816_FREQUENCY_1000000</code> , <code>ISO7816_FREQUENCY_1250000</code> , <code>ISO7816_FREQUENCY_1875000</code> , <code>ISO7816_FREQUENCY_2500000</code> , <code>ISO7816_FREQUENCY_3750000</code> , <code>ISO7816_FREQUENCY_5000000</code> , <code>ISO7816_FREQUENCY_7500000</code> or <code>ISO7816_FREQUENCY_15000000</code> .
<code>uint16_t</code> F	16	Specify a non-ISO value for F.
<code>uint16_t</code> D	16	Specify a non-ISO value for D.
<code>union</code> TProtocolData ProtocolData	56	See definition of TProtocolData for details.

Table 21.3: Definition of TISO7816CommSettings

Members	Length (Bits)	Description
TProtocolDataT0 T0	40	See definition of TProtocolDataT0 for details.
TProtocolDataT1 T1	56	See definition of TProtocolDataT1 for details.

Table 21.4: Definition of TProtocolData

Members	Length (Bits)	Description
<code>byte</code> <code>bmFindexDindex</code>	8	Bit 7-4: FI, Index into table 7 of ISO/IEC 7816-3:2006 selecting a clock rate conversion factor. Bit 3-0: DI, Index into table 8 of ISO/IEC 7816-3:2006 selecting a baud rate conversion factor. This value shall be obtained from TA1 of the ATR.
<code>byte</code> <code>bmTCCKST0</code>	8	This value shall be set to 00h.
<code>byte</code> <code>bGuardTimeT0</code>	8	Extra Guardtime between two characters. Add 0 to 254 etu to the normal guardtime of 12 etu. FFh is the same as 00h. This value shall be obtained from TC1 of the ATR.
<code>byte</code> <code>bWaitingIntegerT0</code>	8	Waiting time between transmission of a command and reception of the response. This value is specified in TC2 of the ATR. If TC2 is not present, the default value is 10.
<code>byte</code> <code>bClockStop</code>	8	This value shall be set to 00h.

Table 21.5: Definition of TProtocolDataT0

## 21.5 Transparent Data Transmission

This function shall be used for byte-wise communication with the card.

```
bool ISO7816_Transceive
(
  int Channel,
  const byte* TX,
  int LenTX,
  byte* RX,
  int* LenRX,
  int MaxRXByteCnt
);
```

Members	Length (Bits)	Description
<code>byte</code> <code>bmFindexDindex</code>	8	Bit 7-4: FI, Index into table 7 of ISO/IEC 7816-3:2006 selecting a clock rate conversion factor. Bit 3-0: DI, Index into table 8 of ISO/IEC 7816-3:2006 selecting a baud rate conversion factor. This value shall be obtained from TA1 of the ATR.
<code>byte</code> <code>bmTCKST1</code>	8	This value shall be set to 00h.
<code>byte</code> <code>bGuardTimeT1</code>	8	Extra Guardtime (0 to 254 etu between two characters). If value is FFh, then guardtime is reduced by 1 etu. This value shall be obtained from TC1 of the ATR.
<code>byte</code> <code>bWaitingIntegerT1</code>	8	Bit 7-4: BWI, values 0-9 valid. Bit 3-0: CWI, values 0-Fh valid. This value is specified in the first TB for T=1 in the ATR.
<code>byte</code> <code>bClockStop</code>	8	This value shall be set to 00h.
<code>byte</code> <code>bIFSC</code>	8	Size of negotiated IFSC in bytes. This value is specified in the first TA for T=1 in the ATR.
<code>byte</code> <code>bNadValue</code>	8	This value shall be set to 00h.

Table 21.6: Definition of TProtocolDataT1

Parameters:

<code>int</code> Channel	Specify a communication channel by this parameter. Valid values are CHANNEL_SAM1 through CHANNEL_SAM4 or CHANNEL_SC1, use one of these predefined constants.
<code>const byte*</code> TX	This buffer holds the data which shall be transmitted to the card.
<code>int</code> LenTX	This parameter specifies the data-length in bytes which shall be transmitted to the card.
<code>byte*</code> RX	This buffer holds the data which was read from the card. Take care for adequate dimensioning.
<code>int*</code> LenRX	After successful completion of this function, this parameter holds the number of bytes read from the card.
<code>int</code> MaxRXByteCnt	This parameter holds the array-size of RX in bytes.
<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> .

## 21.6 Exchange Of APDUs

This function shall be used for APDU exchange based on T=0/T=1 protocol according to ISO7816-3.

```
bool ISO7816_ExchangeAPDU
(
    int Channel,
    const TISO7816_ProtocolHeader* Header,
    const byte* TXData,
    int TXByteCnt,
    byte* RXData,
    int* RXByteCnt,
    int MaxRXByteCnt,
    uint16_t* StatusWord
);
```

Members	Length (Bits)	Description
<code>byte</code> CLA	8	This member holds the CLA-value.
<code>byte</code> INS	8	This member holds the INS-code.
<code>byte</code> P1	8	This member holds the parameter P1.
<code>byte</code> P2	8	This member holds the parameter P2.
<code>uint16_t</code> Lc	16	This member holds Lc which defines the size of the following data-field.
<code>uint16_t</code> Le	16	This member holds Le which defines the maximum expected size of the response data-field.
<code>struct</code> TIS07816_ProtocolHeaderFlags Flags	8	This member holds additional APDU information.

Table 21.7: Definition of TIS07816\_ProtocolHeader

Parameters:

<code>int</code> Channel	Specify a communication channel by this parameter. Valid values are CHANNEL_SAM1 through CHANNEL_SAM4 or CHANNEL_SC1, use one of these predefined constants.
<code>const</code> TIS07816_ProtocolHeader *Header	This parameter holds basic APDU information.
<code>const byte*</code> TXData	This buffer holds the data field of the APDU.
<code>int</code> TXByteCnt	This parameter specifies the data-length in bytes of the data-field.
<code>byte*</code> RXData	This buffer holds the data-field of the received APDU.
<code>int*</code> RXByteCnt	After successful completion of this function, this parameters holds the data-field size of the received APDU.
<code>int</code> MaxRXByteCnt	This parameter holds the array-size of RXData in bytes.
<code>uint16_t*</code> StatusWord	This parameter holds the status word received from the card.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`.

Members	Length (Bits)	Description
<code>byte</code> LePresent	1	If set to true, Le is transmitted.
<code>byte</code> ExtendedAPDU	1	If set to true, this APDU is sent as Extended APDU.
<code>byte</code> RFU	6	Reserved for future use.

Table 21.8: Definition of TISO7816\_ProtocolHeaderFlags

## 21.7 Examples

### 21.7.1 PPS Example

The following example shows how to make a PPS with an ISO7816 card.

```
bool ISO7816_PPS(int Channel, byte Protocol, byte* bmFindexDindex)
{
    byte Cmd[4];
    byte Res[4];
    int TxByteCnt;
    int RxByteCnt;

    // PPS always starts with 0xFF
    Cmd[0] = 0xFF;
    // The second byte stores the desired protocol
    Cmd[1] = Protocol & 0x0F;
    // Is bmFindexDindex present?
    if (bmFindexDindex != NULL)
    {
        // Yes, prepare the command accordingly
        Cmd[1] |= 0x10;
        Cmd[2] = *bmFindexDindex;
        // Calculate the BCC over all command bytes
        Cmd[3] = Cmd[0] ^ Cmd[1] ^ Cmd[2];
        TxByteCnt = 4;
    }
    else
    {
        // FindexDindex is not present, calculate only BCC
        Cmd[2] = Cmd[0] ^ Cmd[1];
        TxByteCnt = 3;
    }
    // Send PPS request to the card, get response
    if (!ISO7816_Transceive(Channel, Cmd, TxByteCnt, Res,
                           &RxByteCnt, sizeof(Res)))
        return false;
}
```

```

    // We expect the card to echo the request in its response
    if (RxByteCnt != TxByteCnt)
        return false;
    return memcmp(Cmd, Res, RxByteCnt) == 0;
}

```

### 21.7.2 Communication Example

The following example shows how to prepare a ISO7816 card for communication at T=1 protocol and exchange APDUs.

```

byte ATR[32];
int ATRByteCnt;

TISO7816SlotStatus SlotStatus;
TProtocolDataT1 ProtocolDataT1;

TISO7816_ProtocolHeader Header;
byte TXData[128];
byte RXData[128];
int RXByteCnt;
uint16_t SW12;

// We want to use T=1 protocol with the following non-default values
ProtocolDataT1.bmFindexDindex = 0x98;
ProtocolDataT1.bmTCCKST1 = 0;
ProtocolDataT1.bGuardTimeT1 = 0xFF;
ProtocolDataT1.bmWaitingIntegersT1 = 0x55;
ProtocolDataT1.bClockStop = 0;
ProtocolDataT1.bIFSC = 0xFE;
ProtocolDataT1.bNadValue = 0x00;

MainLoop:
while (true)
{
    // Is a card inserted in CHANNEL_SC1?
    if (!ISO7816_GetSlotStatus(CHANNEL_SC1, &SlotStatus))
        goto MainLoop;
    // Card slot empty?
    if (SlotStatus.bStatus.bmICCStatus == ISO7816_NOICCPRESENT)
        goto MainLoop;
    // Perform activation of the card and receive ATR
    if (!ISO7816_IccPowerOn
        (
            CHANNEL_SC1,
            ATR,
            &ATRByteCnt,
            sizeof(ATR),

```



```
        ISO7816_POWERSELECT_5V,  
        &SlotStatus.bStatus,  
        &SlotStatus.bError  
    ))  
    goto MainLoop;  
// We expect the card to be present and active  
if (SlotStatus.bStatus.bmICCStatus != ISO7816_ICCPRESENTANDACTIVE)  
    goto MainLoop;  
// Perform PPS for T=1 protocol  
if (!ISO7816_PPS(CHANNEL_SC1, ISO7816_PROTOCOL_T1,  
                &ProtocolDataT1.bmFindexDindex))  
    goto MainLoop;  
  
// Let's prepare our APDU. We want to select the Masterfile (MF)  
// of a PKI card by its SFI (0x3F00).  
Header.CLA = 0x00;  
Header.INS = 0xA4;  
Header.P1  = 0x00;  
Header.P2  = 0x00;  
Header.Lc  = 0x0002;  
Header.Le  = 0x0000;  
Header.Flags.LePresent = true;  
Header.Flags.ExtendedAPDU = false;  
TXData[0]  = 0x3F;  
TXData[1]  = 0x00;  
  
// Exchange the APDU  
if (!ISO7816_ExchangeAPDU(CHANNEL_SC1, &Header, TXData, Header.Lc,  
                          RXData, &RXByteCnt, sizeof(RXData), &SW12))  
    goto MainLoop;  
// Check status word of the received APDU  
if (SW12 == 0x9000)  
{  
    // Further APDUs may follow...  
    DoSomething();  
}  
}
```

## 22 Cryptographic Operations

The cryptographic API incorporates methods for encryption/decryption of data, these are Triple-DES (Data Encryption Standard) or AES (Advanced Encryption Standard). TDES is available in two versions that support different key-lengths: 128 bit (TDES2K) and 192 bit (TDES3K).

The implementation of TDES is based on FIPS PUB 46-3. The method always operates on entire data blocks of 8 bytes. The DES algorithm is passed three times for one TDES operation. In case of TDES2K, the 128 bit key is hereby split into two parts: K1 and K2. In case of TDES3K, the 192 bit key is split into three parts: K1, K2 and K3.

The implementation of AES is based on FIPS PUB 197. The method always operates on entire data blocks of 16 bytes, the key-length is 128 bit.

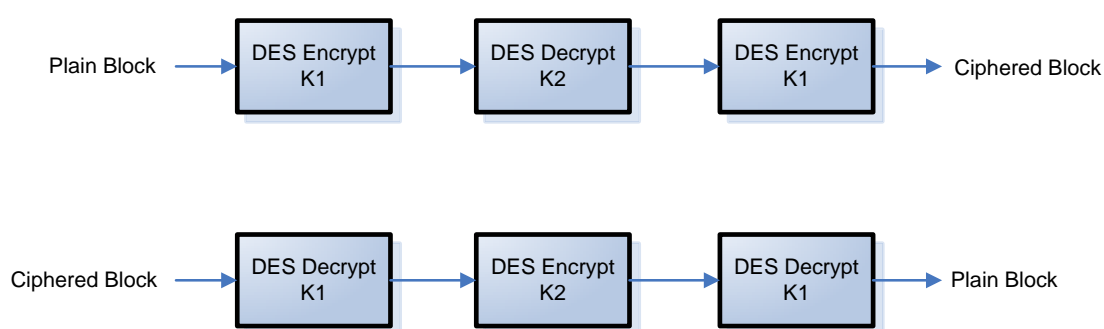


Figure 22.1: TDES2K Operation

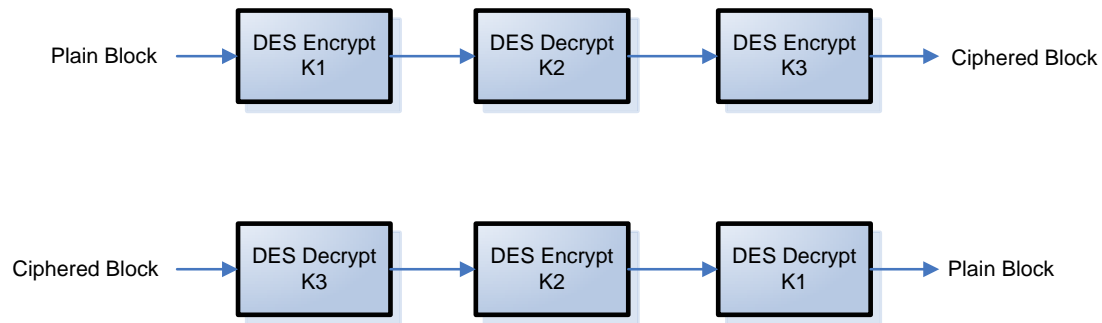


Figure 22.2: TDES3K Operation

The cryptographic API may be used to simply encrypt/decrypt a single block or to encrypt/decrypt a chain of blocks using the CBC-method (Ciphered Block Chaining).

In CBC mode, every ciphering operation depends on the foregoing step, this is achieved by involving the so-called Init Vector IV. The first CBC-operation usually works with an Init Vector that is set to zero.

For encryption, a plain data block P is logically XOR-ed with this Init Vector before it comes to encryption. The result is a ciphered block C which serves as Init Vector for the next operation. See the schematic below for details:

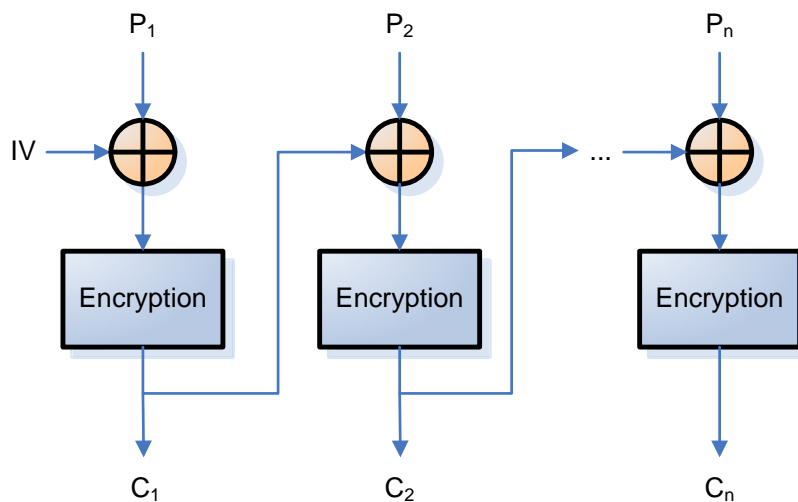


Figure 22.3: CBC Enciphering scheme

If a ciphered block  $C$  is decrypted, the result is logically XOR-ed with the Init Vector. See the schematic below for details:

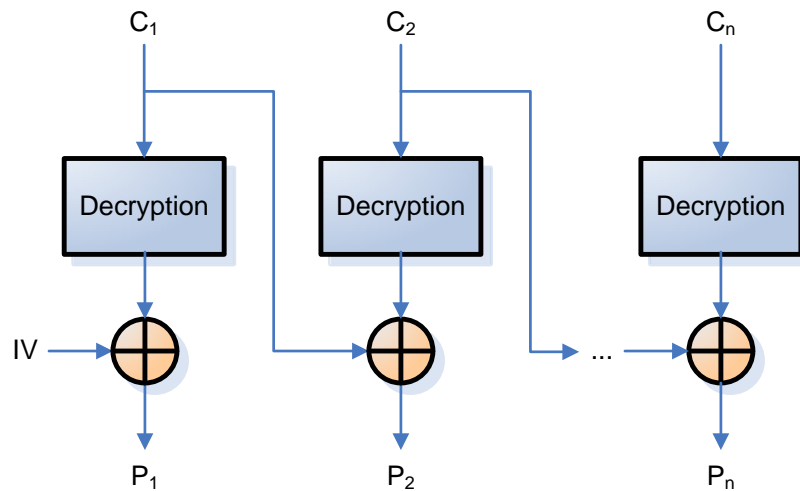


Figure 22.4: CBC Deciphering scheme

## 22.1 Initialization

The cryptographic API has to be initialized before it can be used. During initialization the key is passed to the cryptographic method and assigned to a cryptographic environment. After initialization the functions for encryption and decryption are set up for the desired cryptographic mode. If a cryptographic environment is configured for CBC-operation, the internally managed Init Vector is automatically reset to zero.

```
void Crypto_Init  
(  
    int CryptoEnv,  
    int CryptoMode,  
    const byte* Key,  
    int KeyByteCnt  
);
```

Parameters:

<code>int</code> CryptoEnv	Specify a cryptographic environment by this parameter. The valid range is CRYPTO_ENV0 to CRYPTO_ENV3, use one of these predefined constants.
<code>int</code> CryptoMode	Specify the mode of cryptographic operation. Choose either one of the predefined non-CBC constants CRYPTOMODE_3DES, CRYPTOMODE_3K3DES, CRYPTOMODE_AES128 or one of the pre-defined CBC constants CRYPTOMODE_CBC_DES, CRYPTOMODE_CBC_DFN_DES, CRYPTOMODE_CBC_3DES, CRYPTOMODE_CBC_DFN_3DES, CRYPTOMODE_CBC_3K3DES, CRYPTOMODE_CBC_AES128.
<code>const byte*</code> Key	The key is passed by this parameter. Depending on the specified crypto mode, the key-length is either 16 or 24 bytes.
<code>int</code> KeyByteCnt	Specify the length of the key in bytes.
<u>Return:</u>	This function has no return value.

## 22.2 Encrypt

Use this function to encrypt a plain block of data.

```
void Encrypt
(
    int CryptoEnv,
    const byte* PlainBlock,
    byte* CipheredBlock,
    int BlockByteCnt
);
```

Parameters:

<code>int</code> CryptoEnv	Specify a cryptographic environment by this parameter. The valid range is CRYPTO_ENV0 to CRYPTO_ENV3, use one of these predefined constants.
<code>const byte*</code> PlainBlock	Pointer to the array, that contains the plain data block to be encrypted.
<code>byte*</code> CipheredBlock	Pointer to the array, that receives the encrypted data block. Take care for proper dimensioning.
<code>int</code> BlockByteCnt	Specify the number of bytes of a block.
<u>Return:</u>	This function has no return value.

## 22.3 Decrypt

Use this function to decrypt an encrypted block of data.

```
void Decrypt
(
    int CryptoEnv,
    const byte* CipheredBlock,
    byte* PlainBlock,
    int BlockByteCnt
);
```

### Parameters:

<code>int CryptoEnv</code>	Specify a cryptographic environment by this parameter. The valid range is CRYPTO_ENV0 to CRYPTO_ENV3, use one of these predefined constants.
<code>const byte* CipheredBlock</code>	Pointer to the array, that holds the encrypted data block.
<code>const byte* PlainBlock</code>	Pointer to the array, that receives the decrypted data block. Take care for proper dimensioning.
<code>int BlockByteCnt</code>	Specify the number of bytes of a block.

Return: This function has no return value.

## 22.4 Reset Init Vector

Use this function to manually reset the internally managed Init Vector of a cryptographic environment to zero.

```
void CBC_ResetInitVector
(
    int CryptoEnv
);
```

### Parameters:

<code>int CryptoEnv</code>	Specify a cryptographic environment by this parameter. The valid range is CRYPTO_ENV0 to CRYPTO_ENV3, use one of these predefined constants.
----------------------------	--

Return: This function has no return value.

## 23 Storage Functions

This chapter describes functions for accessing the storage of TWN4.

The storage memory is part of the internal flash of the main control unit (MCU) of TWN4. The gross amount of this storage is 48kByte. Due to segmentation of the memory and further control mechanisms, after deduction the memory size is 18kByte.

Before first use of the storage, the storage must be formatted. The appropriate system function for doing so is `FSFormat`.

In order to gain access to the storage memory, the file system must be initialized and connected to the internal flash. This can be achieved with the system function `FSMount`.

Why is a separate mount needed to gain access to the storage memory?

The reason for a separate mount is, that there could be a reasonable amount of time required in order to start the file system. Background is, that depending on the state of the file system, additional activities must be started, before access of the storage memory is possible. There is especially the situation, which can occur, if last file operation were interrupted by a unplanned power fail. This can lead to the situation, that the file system must be reset to the state, before the interrupted file operation was started. This clean-up is done by function `FSMount`.

The structure of the storage memory is similar to a none-hierarchical file system. Following points must be known:

- Data is structured in files.
- Files are indicated by a file ID. The file ID is any 32 bit number.
- It is possible to iterate through the existing files and thus list the files stored in the memory.
- There is a maximum number of files, which can be stored in the memory. This maximum number is 16.
- In order to read from or write to files, appropriate system functions are available. In order to start a file operation, the file must be opened for appropriate file operation. The maximum number of files, which can be kept opened at a time is 4.
- File operations are kept atomic. This means: If a change to a file (some kind write operation) is interrupted by a power fail, the file system returns to the state, where the change began.



## 23.1 Management Functions

### 23.1.1 FSMount

Before any access to files can be performed, the appropriate file system must be mounted. Following steps are performed by function FSMount:

- Check, if the specified volume contains a valid file system.
- Check, if there is a not completed file operation.
- If applicable, unwind file system to the point where not completed file operation was started.
- Finally, create a logical link between volume and file system.

```
bool FSMount(int StorageID, int Mode)
```

#### Parameters:

int StorageID	Specifies the volume, which should be mounted. Currently, there is one storage available, the internal flash. The appropriate definition for this storage is SID_INTERNALFLASH
int Mode	Specifies the mode in which the volume is mounted. This can be FS_MOUNT_NONE (equivalent to a unmount), FS_MOUNT_READONLY (no write access to storage possible) or FS_MOUNT_READWRITE (full read/write access).

Return: If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

### 23.1.2 FSFormat

This function prepares the storage memory of TWN4 for further file operations.

— WARNING — WARNING — WARNING —

All data, which is stored on the file system will be irrecoverable deleted by calling this function!

```
bool FSFormat(int StorageID, int MagicValue)
```

Parameters:

<code>int StorageID</code>	Specifies the volume, which should be formatted.
<code>int MagicValue</code>	In order to avoid accidentally format of a volume, an appropriate parameter for MagicValue must be specified. There is a definition for this magic value, which is <code>FS_FORMATMAGICVALUE</code> .

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

## 23.2 File Functions

### 23.2.1 FSOpen

This function must be called in order to begin any read or write operation from/to a file.

Following definitions for the parameter mode are valid:

<code>FS_READ</code>	Open a file for read access. If the file not exists, an error is generated. The position of the read pointer is set to zero, thus to the start of the file.
<code>FS_WRITE</code>	Open a file for write access. An empty file is created independently of if the file already exists or not, thus content of an earlier version of that file will be deleted.
<code>FS_APPEND</code>	Open a file for write access. If the file does not exist, a new file is created. If the file already exists, the file pointer is moved to the end of the file, which means, that newly written data is appended to data of existing file.

Following further considerations:

- A file can be opened one time in mode `FS_WRITE` or `FS_APPEND`, but never, if it is already opened by any other file operation.
- A file can be opened many times in mode `FS_READ`, but never, if it is already opened in mode `FS_WRITE` or `FS_APPEND` by another file operation.

```
bool FSOpen(int FileEnv, int StorageID, uint32_t FileID, int Mode)
```

Parameters:

<code>int FileEnv</code>	Specifies the environment to be used for the file operation. Up to four file operations can be opened at a time. The appropriate definitions for these environments are <code>FILE_ENV0</code> - <code>FILE_ENV3</code> .
<code>int StorageID</code>	Specifies the storage on which the file resides. Currently, this parameter can be <code>SID_INTERNALFLASH</code> only.
<code>uint32_t FileID</code>	Specifies the ID of a file. The file ID is a reduced version of file name and be understood as such. File ID is an integer number from 1 to $2^{32} - 1$ , thus <code>0x00000000</code> to <code>0xFFFFFFFF</code> .
<code>int Mode</code>	Specifies, how the file is accessed (see above).

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

X

**23.2.2 FSClose**

This function is used to terminate a file operation. Several actions are taken, when this function is called:

- Pending data is written to the storage system.
- If this is the last file being closed, the file system is finalized in terms, that the even loss of the power will restore this now achieved state.

```
bool FSClose(int FileEnv)
```

Parameters:

<code>int FileEnv</code>	Specifies the environment to be used for the file operation.
--------------------------	--

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

**23.2.3 FSCloseAll**

This function is closing all opened file operations throughout all mounted storages. This function avoids keeping track of opened file operations.

```
void FSCloseAll(void)
```

<u>Parameters:</u>	None.
<u>Return:</u>	None.

### 23.2.4 FSeek

Read and write operations from/to a file are implemented via a file pointer, which references the point, from which next data is read or where next data is written. With this function, the file pointer can be moved throughout a file and furthermore in relation to a specific point of the file.

FS_POSABS	Move file position in relation to the start of the file. This results in a move of the file pointer to an absolute position.
FS_POSREL	Move the file pointer in relation to the current position. This allows an easy skip of a number of bytes of the file.
FS_POSEND	Move the file pointer in relation to the end of the file. This allows to move to the end of the file without knowledge and independent of the length of a file.

```
bool FSeek(int FileEnv, int Origin, int Pos)
```

Parameters:

int FileEnv	Specifies the environment to be used for the file operation.
int Origin	Specifies the reference point, from which the new file position is calculated (see above).
int Pos	Specifies the number of bytes in relation to the reference point. A negative value is treated as position before reference point, a positive value is treated as position behind the reference point.

<u>Return:</u>	If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.
----------------	---

### 23.2.5 FTell

This function returns the position of the file pointer in relation to a reference point. Please note that in consequence, specifying FS\_POSREL as origin must always return the value zero.

```
bool FTell(int FileEnv, int Origin, int *Pos)
```

Parameters:

<code>int FileEnv</code>	Specifies the environment to be used for the file operation.
<code>int Origin</code>	Specifies the reference point, under which the current position is calculated (see function <code>FSSeek</code> ).
<code>int *Pos</code>	A pointer to an integer, which will receive the value of the position.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

### 23.2.6 FSReadBytes

Read bytes from a file, which has been opened in mode `FS_READ` before. Use function `FSOpen` to open the file accordingly.

The function generates the error `ERR_ENDOFFILE`, if less than the requested number of bytes were read from the file or if there are no more bytes left to be read from the file.

```
bool FSReadBytes(int FileEnv, void *Data, int ByteCount, int *BytesRead)
```

Parameters:

<code>int FileEnv</code>	Specifies the environment to be used for the file operation.
<code>void *Data</code>	Pointer to an array of bytes, which receives read data.
<code>int ByteCount</code>	Number of bytes, which should be read from the file.
<code>int *BytesRead</code>	Pointer to an integer, which receives the number of actually read bytes. The received value is valid even if the function returns with an error.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

### 23.2.7 FSWriteBytes

Write bytes to a file, which has been opened in mode `FS_WRITE` or `FS_APPEND` before. Use function `FSOpen` to open the file accordingly.

```
bool FSWriteBytes(int FileEnv, const void *Data, int ByteCount, int *BytesWritten)
```

Parameters:

<code>int FileEnv</code>	Specifies the environment to be used for the file operation.
<code>const void *Data</code>	Pointer to an array of bytes, which contains data to be written.
<code>int ByteCount</code>	Number of bytes, which should be written to the file.
<code>int *BytesWritten</code>	Pointer to an integer, which receives the number of actually written bytes. The received value is valid even if the function returns with an error.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. A concrete error code can be retrieved with system function `GetLastError`.

## 23.3 Directory Functions

### 23.3.1 FSFindFirst

The functions `FSFindFirst/FSFindNext` implement the possibility to enumerate the files contained in a files system. In order to begin enumeration of files the function `FSFindFirst` must be called.

The members of a directory entry are stored in a structure of type `TFileInfo`. The members of the structure are:

<code>ID</code>	The file ID.
<code>Length</code>	The length of the file.

```
bool FSFindFirst(int StorageID, TFileInfo *pFileInfo)
```

Parameters:

<code>int StorageID</code>	Storage ID of the file system, where files should be enumerated.
<code>TFileInfo *pFileInfo</code>	Pointer to a structure of type <code>TFileInfo</code> which receives a directory entry.

Return:

If the operation was successful, the return value is `true`, otherwise it is `false`. If no directory entry was found the error code `ERR_FILENOTFOUND` is generated. The concrete error code can be retrieved with system function `GetLastError`.

### 23.3.2 FSFindNext

The functions FSFindFirst/FSFindNext implement the possibility to enumerate the files contained in a files system. In order to continue enumeration, once first entry has been retrieved with function FSFindFirst, the function FSFindNext must be called.

```
bool FSFindNext(TFileInfo *pFileInfo)
```

Parameters:

TFileInfo \*pFileInfo    Pointer to a structure of type TFileInfo which receives a directory entry.

Return:

If the operation was successful, the return value is true, otherwise it is false. If no directory entry was found the error code ERR\_FILENOTFOUND is generated. The concrete error code can be retrieved with system function GetLastError.

### 23.3.3 FSDelete

Use function FSDelete to delete files from the file system. A file, which is currently opened can not be deleted.

```
bool FSDelete(int StorageID,uint32_t FileID)
```

Parameters:

int StorageID            Storage ID of the file in question.

uint32\_t FileID          File ID of the file to be deleted.

Return:

If the operation was successful, the return value is true, otherwise it is false. A concrete error code can be retrieved with system function GetLastError.

### 23.3.4 FSRename

Use function FSRename to rename files on the file system.

```
bool FSRename(int StorageID,uint32_t OldFileID,uint32_t NewFileID)
```

Parameters:

<code>int StorageID</code>	Storage ID of the file in question.
<code>uint32_t OldFileID</code>	Current file ID of the file to be renamed.
<code>uint32_t NewFileID</code>	Future file ID of the file to be renamed.
<u>Return:</u>	If the operation was successful, the return value is <code>true</code> , otherwise it is <code>false</code> . A concrete error code can be retrieved with system function <code>GetLastError</code> .

**23.3.5 Examples**

The following example shows a minimal App which is storing data (a structure with constant size) in the file system.

Here is an example for a function, which reads a complete file from the file system. The file system must have been mounted before with function `FSMount`.

```
bool ReadFile1(uint32_t FileID, byte *Data, int *FileLength, int MaxFileLength)
{
    if (!FSOpen(FILE_ENV0, SID_INTERNALFLASH, FileID, FS_READ))
        return false;
    FSReadBytes(FILE_ENV0, Data, MaxFileLength, FileLength);
    int LastError = GetLastError();
    FSClose(FILE_ENV0);
    if (LastError != ERR_NONE && LastError != ERR_ENDOFFILE)
        return false;
    // Function was successfully completed
    return true;
}
```

Here is an example for a function, which reads a complete file from the file system but in portions of 256 bytes. This might be useful, if the implementation is actually done on a host, which is doing system calls indirectly via TWN4 Simple Protocol. The file system must have been mounted before with function `FSMount`.

```
bool ReadFile2(uint32_t FileID, byte *Data, int *Length, int ExpectedLength)
{
    *Length = 0;
    if (!FSOpen(FILE_ENV0, SID_INTERNALFLASH, FileID, FS_READ))
        return false;
    bool ReadSuccess;
    int RemainingBytes = ExpectedLength;
    do
    {
        if (RemainingBytes == 0)
        {
            FSClose(FILE_ENV0);
        }
    }
}
```



```

        return true;
    }
    const int BlockSize = 256;
    int BytesToRead = RemainingBytes;
    if (BytesToRead > BlockSize)
        BytesToRead = BlockSize;
    int BytesRead;
    ReadSuccess = FSReadBytes(FILE_ENVO,Data,BytesToRead,&BytesRead);
    Data += BytesRead;
    *Length += BytesRead;
    RemainingBytes -= BytesRead;
}
while (ReadSuccess);
int LastError = GetLastError();
FSClose(FILE_ENVO);
if (LastError != ERR_NONE && LastError != ERR_ENDOFFILE)
    return false;
// Function was successfully completed
return true;
}

```

Here is an example for a function, which writes a complete file to the file system in portions of 256 bytes. This might be useful, if the implementation is actually done on a host, which is doing system calls indirectly via TWN4 Simple Protocol. The file system must have been mounted before with function FSMount.

```

bool WriteFile(uint32_t FileID,byte *Data,int Length)
{
    if (!FSOpen(FILE_ENVO,SID_INTERNALFLASH,FileID,FS_WRITE))
        return false;
    bool WriteSuccess;
    int RemainingBytes = Length;
    do
    {
        if (RemainingBytes == 0)
        {
            FSClose(FILE_ENVO);
            return true;
        }
        const int BlockSize = 256;
        int BytesToWrite = RemainingBytes;
        if (BytesToWrite > BlockSize)
            BytesToWrite = BlockSize;
        int BytesWritten;
        WriteSuccess = FSWriteBytes(FILE_ENVO,Data,BytesToWrite,&BytesWritten);
        Data += BytesWritten;
        RemainingBytes -= BytesWritten;
    }
    while (WriteSuccess);
}

```

```
int LastError = GetLastError();
FSClose(FILE_ENV0);
if (LastError != ERR_NONE)
    return false;
// Function was successfully completed
return true;
}
```

## 24 System Parameters

The TWN4 App-system provides methods of setting up parameters before or during runtime of Apps.

- In order to set up parameters before the App is started, a so-called Manifest can be specified as part of an App.
- In order to set up parameters during normal execution of an App there is the system function `SetParameters`.

This section describes the specification of a Manifest and all available parameters. See chapter "System Functions" for a description of function `SetParameters`.

### 24.1 TLV Format

Parameters for a Manifest or the system function `SetParameters` are specified in the TLV format. The TLV format specifies a chain of parameters with variable type and length. This format must follow following rules:

- Every entry (except the last entry) is a sequence of 3 items. The 3 items are 'Type', 'Length' and 'Value'.
- The name of the parameter is associated to 'Type', the length of
- the value is associated to 'Length' and the value itself is associated to 'Value'
- The TLV list must be terminated with an item consisting of just the type. This type must contain the value `TLV_END`.

### 24.2 Manifest

The intention for specifying a Manifest as part of an App could be to avoid opening of communication channels in order to further reduce current consumption. Another could be to modify behaviour of the USB section of TWN4.

The specification of a Manifest is pretty simple:

Define an array of bytes with the key-name Manifest. This will point the firmware of TWN4 to the position where the parameters of interest are stored. Here is an example:

Example:

```
// This sample demonstrates the specification of a Manifest:
const byte *Manifest =
{
    OPEN_PORTS, 1, OPEN_PORT_USB_MSK,    // Open USB channel only
    TLV_END                               // End of TLV
};
```

No further action is required.

## 24.3 Available Parameters

Here is a list of all parameters, which are supported:

Type (Parameter)	Length	Value
TLV_END	N/A	N/A
OPEN_PORTS	1	Bitwise OR of one or more of the following definitions: OPEN_PORT_USB_MSK OPEN_PORT_COM1_MSK OPEN_PORT_COM2_MSK
EXECUTE_APP	1	EXECUTE_APP_AUTO EXECUTE_APP_ALWAYS
INDITAG_READMODE	1	INDITAG_READMODE_1 INDITAG_READMODE_2
COTAG_READMODE	1	COTAG_READMODE_HASH COTAG_READMODE_1 COTAG_READMODE_2
COTAG_VERIFY	1	COTAG_VERIFY_OFF COTAG_VERIFY_ON
HONEYTAG_READMODE	1	HONEYTAG_READMODE_HASH HONEYTAG_READMODE_1
ICLASS_READMODE	1	ICLASS_READMODE_UID ICLASS_READMODE_PAC
AT55_BITRATE	1	8 to 128 as multiple of 2
AT55_OPTIONS	1	Bitwise OR of one or more of the following definitions: AT55_OPT_SEQUENCETERMINATOR_ON AT55_OPT_SEQUENCETERMINATOR_OFF

Continued from last page:

HITAG1S_T0	1	Values from 14 to 40
HITAG1S_T1	1	Values from 14 to 40
HITAG1S_TGAP	1	Values from 2 to 14
HITAG2_T0	1	Values from 14 to 40
HITAG2_T1	1	Values from 14 to 40
HITAG2_TGAP	1	Values from 2 to 14
ISO14443_BITRATE_TX	1	One of the following possible bitrates: ISO14443_BITRATE_106 ISO14443_BITRATE_212 ISO14443_BITRATE_424 ISO14443_BITRATE_848
ISO14443_BITRATE_RX	1	One of the following possible bitrates: ISO14443_BITRATE_106 ISO14443_BITRATE_212 ISO14443_BITRATE_424 ISO14443_BITRATE_848

## 25 System Errors

Here is a list of all error codes, which are generated by the firmware of TWN4. The error codes can be retrieved with function `GetLastError`.

In the current version of the firmware, storage functions (FS...) are generating such errors.

Error Code	Description
ERR_NONE	No error occurred.
ERR_OUTOFMEMORY	The execution of a function required more memory than was available.
ERR_ISALREADYINIT	There was a try to initialize a system module, which already was initialized.
ERR_NOTINIT	There was a try to use a function from a module, which is not initialized.
ERR_ISALREADYOPEN	There was a try to open a system resource, which is already open.
ERR_NOTOPEN	There was a try to use a system resource, which must be opened before usage.
ERR_RANGE	A specified parameters exceeded the valid range of values.
ERR_PARAMETER	A specified parameters is not in set of valid parameters.
ERR_UNKNOWNSTORAGEID	A storage ID was specified, which is not known by the firmware.
ERR_WRONGINDEX	A index was specified, which was out of the valid range.
ERR_FLASHERASE	The erase of a section of the flash failed.
ERR_FLASHWRITE	The write to the flash memory failed.
ERR_SECTORNOTFOUND	A sector of the file system was not found.
ERR_STORAGEFULL	All sectors of the file system are occupied.
ERR_STORAGEINVALID	There is an error in the file system.

ERR_TRANSACTIONLIMIT	The limit of changes in the file system is reached, which is possible within one transactions.
ERR_UNKNOWNFS	The file system on the specified storage is not supported by the current firmware.
ERR_FILENOTFOUND	The specified file was not found.
ERR_FILEALREADYEXISTS	The specified file already exists.
ERR_ENDOFFILE	The end of the file was reached. There is no more data to be read. Note: This error code is generated even the system function returned successful execution.
ERR_STORAGENOTFOUND	The specified storage was not found, e.g. because it is not mounted.
ERR_STORAGEALREADYMOUNTED	The specified storage is already mounted.
ERR_ACCESSDENIED	The access to a file was denied, e.g. write access to a file in a storage, which is mounted as read only.
ERR_FILECORRUPT	The specified file is corrupt in terms of a corrupted file system.
ERR_INVALIDFILEENV	The specified environment is invalid.
ERR_INVALIDFILEID	The specified file ID is invalid.
ERR_RESOURCELIMIT	The maximum number of available resources have been occupied.

Please see file `twm4.sys.h` (which can be found in local directory `Tools\sys\` of the developer pack) for concrete numbers, which are behind the definitions.



## 26 Runtime Library

There is a couple of functions, which are not part of the firmware of TWN4. Instead, they are statically linked to the App.

There are several intentions for such functions:

- Provide functions instead of having similar code in each App.
- Provide an API at a higher level to simplify writing Apps.
- Maintain a degree of compatibility to TWN3.

### 26.1 Timer Functions

Include file: `apptools.h`

There are three functions, which implement a simple API, which allows triggering events after a specified time. The behaviour of the functions are similar to TWN3. Compared to TWN3, there is only one timer available. Therefore no timer ID must be specified. These timer functions are implemented using system function `GetSysTicks`.

#### 26.1.1 StartTimer

Start the timer with a specified time.

```
void StartTimer(unsigned long Duration)
```

Parameters:

`unsigned long` Duration    Time in milliseconds, till function `TestTimer` returns true.

Return:                      None.

#### 26.1.2 StopTimer

Stop the timer, thus function `TestTimer` will never return true.

```
void StopTimer(void);
```

Parameters: None.

Return: None.

### 26.1.3 TestTimer

Test, if the timer reached the timeout which was programmed by function StartTimer.

```
bool TestTimer(void);
```

Parameters: None.

Return: If the timeout has been reached, the function returns true, otherwise, it return false.

## 26.2 Host Communication

Include file: apptools.h

There are several function which implement a simplified interface for direct write to the host. The host is defined to be a communication channel, where all communication takes place. This removes the requirement to specify the communication channel every time when communication should take place.

For a more sophisticated kind of communication (binary, bidirectional), it is suggested to directly use the I/O functions from the firmware.

### 26.2.1 SetHostChannel

Specify the channel, where communication should take place. By default, the channel is determined by the connected communication cable, which is therefore either USB (TWN4 USB) or COM1 (TWN4 RS232).

```
void SetHostChannel(int Channel)
```

Parameters:

`int Channel` Specifies the communication channel to be used. This might be CHANNEL\_USB, CHANNEL\_COM1, CHANNEL\_COM2 or CHANNEL\_I2C or CHANNEL\_NONE. If CHANNEL\_NONE is specified, channel will be choosen depending on connected communication cable.

Return: None.

### 26.2.2 HostTestByte

Use this function to check if there is a byte available in the input buffer of the host-channel.

`bool` HostTestByte(`void`)

Parameters: None.

Return: If there is a byte available, the return value is `true`, otherwise it is `false`.

### 26.2.3 HostReadByte

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

`byte` HostReadByte(`void`)

Parameters: None.

Return: The byte which was read from the input buffer.

### 26.2.4 HostTestChar

Test if a character is available from the host. The character can be read with function HostReadChar.

`bool` HostTestChar(`void`)

Parameters: None.

Return: `true` if at least one character arrived, otherwise `false`.

### 26.2.5 HostReadChar

Receive a character from the host. This is a blocking function. This means, it is waiting, till a character is available.

`char` HostReadChar(`void`)

Parameters: None.

Return: The character, which was read from the host.

### 26.2.6 HostWriteByte

Use this function to send one byte to the host through the actually configured host-channel. If the output buffer is completely occupied, the function blocks until there is enough space.

```
void HostWriteByte(byte Byte)
```

Parameters:

`byte` Byte                      The byte to be sent.

Return:                      None.

### 26.2.7 HostWriteChar

Send a character to the host. This is a blocking function. This means, it is waiting, till there is storage in the output buffer, to transmit the character.

```
void HostWriteChar(char Char)
```

Parameters:

`char` Char                      The character to be sent to the host.

Return:                      None.

### 26.2.8 HostWriteString

Send a string to the host. The string must be terminated with a null character. The string is sent without the null character.

```
void HostWriteString(const char *String)
```

Parameters:

`const char` \*String              Pointer to the string to be sent.

Return:                      None.

### 26.2.9 HostWriteRadix

Send a number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteRadix(const byte *ID,int BitCnt,int DigitCnt,int Radix)
```

Parameters:

<code>const byte *ID</code>	Pointer to the array of bytes.
<code>int BitCnt</code>	Number of bits stored in the array.
<code>int DigitCnt</code>	Number of output digits.
<code>int Radix</code>	Base for conversion from binary to ASCII. Use: <ul style="list-style-type: none"><li>• 2 for binary conversion</li><li>• 8 for octal conversion</li><li>• 10 for decimal conversion</li><li>• 16 for hexadecimal conversion</li></ul>

Return: None.

**26.2.10 HostWriteBin**

Send a binary number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteBin(const byte *ID, int BitCnt, int DigitCnt)
```

Parameters:

<code>const byte *ID</code>	Pointer to the array of bytes.
<code>int BitCnt</code>	Number of bits stored in the array.
<code>int DigitCnt</code>	Number of output digits.

Return: None.

**26.2.11 HostWriteDec**

Send a decimal number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteDec(const byte *ID, int BitCnt, int DigitCnt)
```

Parameters:

<code>const byte *ID</code>	Pointer to the array of bytes.
<code>int BitCnt</code>	Number of bits stored in the array.
<code>int DigitCnt</code>	Number of output digits.

Return: None.

### 26.2.12 HostWriteHex

Send a hexadecimal number to the host in ASCII format. The number is specified by an array of bytes containing the binary data.

```
void HostWriteHex(const byte *ID,int BitCnt,int DigitCnt)
```

Parameters:

<code>const byte *ID</code>	Pointer to the array of bytes.
<code>int BitCnt</code>	Number of bits stored in the array.
<code>DigitCnt</code>	Number of output digits.

Return: None.

### 26.2.13 HostWriteVersion

Send the firmware version to the host. This function is sending the result of function `GetVersionString` to the host.

```
void HostWriteVersion(void)
```

Parameters: None.

Return: None.

## 26.3 Beep Functions

Include file: `apptools.h`

The beep functions implement a simplified API around the system function `Beep`.

### 26.3.1 SetVolume

Set the beeper volume. The default volume is 0.

```
void SetVolume(int NewVolume)
```

Parameters:

<code>int NewVolume</code>	Specify the volume in percent from 0 to 100.
----------------------------	--

Return: None.

### 26.3.2 GetVolume

Read current volume.

```
int GetVolume(void);
```

Parameters: None.

Return: Current volume in arange from 0 to 100.

### 26.3.3 BeepLow

Perform a beep at a frequency of BEEP\_FREQUENCY\_LOW (2057 Hz) with a duration of 50 milliseconds.

```
void BeepLow(void)
```

Parameters: None.

Return: None.

### 26.3.4 BeepHigh

Perform a beep at a frequency of BEEP\_FREQUENCY\_HIGH (2400 Hz) with a duration of 50 milliseconds. This is meant to be the standard signal for a successful operation, e.g. read of a transponder.

```
void BeepHigh(void)
```

Parameters: None.

Return: None.

## 26.4 Compatibility to TWN3

Include file: apptools.h

Currently, there is one function for maintaining 100% backward compatibility to TWN3 applications.

### 26.4.1 ConvertTagTypeToTWN3

This functions converts a tag type from the TWN4 system to TWN3 system. Due to the fact that TWN4 covers a broader range of transponders, the situation might occure, that a

conversion is not possible. Under that circumstance the TWN3 value TAGTYPE\_NONE (0) is returned.

```
int ConvertTagTypeToTWN3(int TagTypeTWN4)
```

Parameters:

`int TagTypeTWN4` Tag type as returned e.g. by TWN4 system function SearchTag.

Return: Corresponding tag type as it would be returned by TWN3 system function TagSearch.

## 26.5 Simple Protocol

Include file: `prs.h`

The Simple Protocol is the standard protocol for building solutions, which need operation of TWN4, which is controlled by the host.

There is a set of functions and definitions, which allow to implement an App, which runs the Simple Protocol. There are some options, which have influence on some details of the Simple Protocol (ASCII/binary mode, CRC). Furthermore, these functions allow to specify custom communication channel and configure the host interface before starting the communication.

The simplest App for using these functions could be written as follows:

```
#include <twon4.sys.h>
#include <prs.h>

int main(void)
{
    InitSimpleProtocol(GetHostChannel(), PRS_COMM_MODE_ASCII | PRS_COMM_CRC_OFF);
    while (true)
    {
        if (SimpleProtoTestCommand())
        {
            SimpleProtoExecuteCommand();
            SimpleProtoSendResponse();
        }
    }
}
```



### 26.5.1 SimpleProtoInit

Use this function to prepare the Simple Protocol for operation. Before starting this function, it is possible to e.g. prepare a serial port with appropriate communication parameters.

```
bool SimpleProtoInit(int Channel, int Mode)
```

Parameters:

`int` Channel                      This parameter specifies the communication channel for the Simple Protocol. This can be one of the channels as defined by the system I/O functions.

`int` Mode                        This parameter specifies the mode of communication. It is a bit-operation, which combines mode (PRS\_COMM\_MODE\_ASCII or PRS\_COMM\_MODE\_BINARY) and CRC (COMM\_CRC\_OFF or PRS\_COMM\_CRC\_ON).

Return:                        This function returns `true`, if initialization was successful. Otherwise it returns `false`.

### 26.5.2 SimpleProtoTestCommand

This is a non-blocking function, which polls for the availability of a command from the host. If the function returns `true`, a command is available. The command is stored in the global variables `SimpleProtoMessage` and `SimpleProtoMessageLength`.

```
bool SimpleProtoTestCommand(void)
```

Parameters:                      None.

Return:                        This function returns `true`, if a command became available. Otherwise it returns `false`.

### 26.5.3 SimpleProtoExecuteCommand

This function executes a command stored in the global variables `SimpleProtoMessage` and `SimpleProtoMessageLength`. After execution of the command, these variables contain the response to be sent to the host.

```
void SimpleProtoExecuteCommand(void)
```

Parameters:                      None.

Return:                        None.

#### 26.5.4 SimpleProtoSendResponse

This function sends a response stored in the global variables SimpleProtoMessage and SimpleProtoMessageLength to the host.

```
void SimpleProtoSendResponse(void)
```

Parameters:                      None.

Return:                              None.

## **27 Compatibility of TWN4 Mini Reader MIFARE NFC**

Due to reduced functionality of TWN4 Mini Reader MIFARE NFC, several API functions are not available. If an API function is called, which is not supported by TWN4 Mini Reader MIFARE NFC the device stops execution of the App and enters exception state (diagnostic LED is flashing three times).

API	Supported	Remark
System Functions	Yes	
I/O Functions	Yes	COM2 and USB are not supported
Memory Functions	Yes	
Peripheral Functions	Yes	Support of GPIO0 to GPIO3 only, Beep is doing delay only
Conversion Functions	Yes	
I2C Functions	No	
RF Functions	Yes	
HITAG 1 and HITAG S Functions	No	
HITAG 2 Functions	No	
EM4x50 Functions	No	
AT55xx Functions	No	
TILF (TIRIS) Functions	No	
LEGIC Functions	No	
MIFARE Classic Functions	Yes	
MIFARE Ultralight (-C) Functions	Yes	
ISO15693 Functions	Yes	
Cryptographic Functions	Yes	
DESFire Functions	Yes	
Contact Card Functions	Yes	SAM1 only
iCLASS Functions	Yes	
ISO14443 Functions	Yes	
NFC SNEP Functions	Yes	
System Parameters	Yes	
Runtime Library	Yes	