

SPIC C Library

User's Guide

Byte Paradigm – info@byteparadigm.com

Table of Content

Table of Content 2
Table of Tables 3
Table of Pictures ..... **Error! Bookmark not defined.**
References 4
History 4
1 Introduction 5
2 SPI Operating Mode 6
2.1 Features 6
2.2 SPI Signals 6
2.3 Signals Mapping on the device connector 7
3 SPIC Library 8
3.1 Functions quick Reference Table 8
3.2 Functions details 10
SPIIC(void) 10
void Terminate(void) 10
bool IsDeviceReady(void) 10
void SetClockCont(bool Cont) 11
bool GetClockCont(void) 11
void SetWrEn (bool High) 11
bool GetWrEn (void) 11
void SetMode(int Mode) 12
int GetMode(void) 12
int SetReqClock(int Freq) 12
int GetReqClock(void) 12
int GetSynthClock(void) 13
void SetNrOfSlaves(int NrSlaves) 13
int GetNrOfSlaves(void) 13
void SelectSlave (int SlaveID) 13
int GetSelectedSlave (void) 13
int SetSSEdges(int SSDelayStart, int SSDelayStop) 13
int GetSSDelayStart(void) 13
void GetSSDelayStop(void) 14
void SetSSActiveLevel (bool Level) 14
bool GetSSActiveLevel (void) 14
void SetSSClockMasking (bool Enable) 14
bool GetSSClockMasking (void) 14
void SetLatencyClockMasking (bool Enable) 14
bool GetLatencyClockMasking (void) 14
void SetBitOrder(bool BitOrder) 14
bool GetBitOrder(void) 14
int WriteConfFile(char *FileName, bool Message) 15
int ReadConfFile(char *FileName, bool Message) 15
int Idle(int NrBits, bool Message) 15
int ShiftWr(int NrBits, void *pDataOut, bool Message = true) 15
int ShiftWrAndRd(int NrBits, void *pDataOut, void *pDataIn, bool Message = true) 16
int ShiftWrThenRd(int NrBitsOut, int NrBitsIn, void *pDataOut, void *pDataIn, int Latency= 1, bool WREnHigh = true, bool Message = true) 17
void InitBurst(void) 17
int IdleBurst(int NrBits, bool Message) 17
int ShiftWrBurst(int NrBits, void *pDataOut, bool Message = true) 18
int ShiftWrAndRdBurst(int NrBits, void *pDataOut, void *pDataIn, bool Message = true) 18

int ShiftWrThenRdBurst(int NrBitsOut, int NrBitsIn, void *pDataOut, void *pDataIn, int Latency = 1, bool WREnHigh = true, bool Message = true) 19
void SendBurst(void) 19
int GetFirstBurstTransfer(void) 19
int GetNextBurstTransfer(void) 20
int GetBurstData(int nr, char *pData) 20
int GetCapturedData(int NrBytes, void *pData) 20
int GetScriptLength (char * FileName) 20
int ExecuteScript (char *FileName) 20
int GetScriptPos (void) 20
int Analyse (unsigned int NrBits, ... 21
bool SPI4Mode, ... 21
unsigned int SPI3WrLength, ... 21
unsigned int SPI3Latency, ... 21
unsigned int SPI3RdLength, ... 21
bool Message) 21
void Abort (void) 21
int GetCurrPos (void) 21
void SetExportFileName (CString FileName) 21
char *GetExportFileName (void) 21
void SetExportFileType (unsigned int FileType) 21
int GetExportFileType (void) 22
void SetAutoSave (bool AutoSave) 22
bool GetAutoSave (void) 22
int ExportRawDataFile (char *FileName) 22
int ExportRawSPIDataFile (char *FileName) 22
int ExportDecodedSPIDataFile (char *FileName) 22
int GetLastErr (void) 23
void SetInternalTrigger (bool Internal) 23
bool GetInternalTrigger (void) 23
void SetEdgeTrigger (bool Enable) 23
bool GetEdgeTrigger (void) 23
void SetCtrlTrigMask (short *pMask) 23
void GetCtrlTrigMask (short *pCtrlTrigMask) 23
void SetCtrlTrigPattern(short *pPattern) 24
void GetCtrlTrigPattern (short *pCtrlTrigPattern) 24
int SetTriggerPos (int Sample) 24
int GetTriggerPos (void) 24
int GetOverSampling (void) 24

Table of Tables

Table 1: SPI signals description 6
Table 2: Quick reference table of ADWG procedures (by functionality) 8

References

[1] GP-22050 data sheet (ds_GP22050.pdf)
This document contains all the technical characteristics of the GP-22050 device.
[2] 8PI Control Panel – SPI mode of operation user's guide (ug_8PIControlPanel_SPI.pdf).

History

Version	Date	Description
1.00	07-Jun-2006	Initial revision
1.01	17-Nov-2006	Updated command list table
1.02	15-Dec-2006	Review for SPI mode of operation update
1.03	16-May-2007	Added the SPI analyser functions
1.04	30-May-2007	Review for release
1.05	05-Sep-2007	Modified SetReqClock and SetSSEdgesDescription
1.06	13-Nov-2007	Update for SPI Xpress
1.07	08-Oct-2008	Added IdleBurst function
1.08	24-Sep-2009	Update for release 1.07a

1 Introduction

The SPIC library is a specialised C library used with the GP-22050 device in SPI mode of operation and for the SPI Xpress device. It provides a set of 'pure C' functions to configure and control the chosen device from within a C/C++ compatible environment. As opposed to the corresponding C++ libraries, this library offers a 'pure C' interface with each function, which is often easier to integrate from within any external environment.

This library calls itself other libraries and functions to manage the low level transfer of data between the host PC and the device. Schematically, any session using the SPIC library starts by connecting itself to the *8PI Smart Router*© application delivered with the *8PI Control Panel*. This application manages the different client connections to the device and handles priorities between the processes and applications. On the other side, it is responsible for the actual data transfers onto the USB connection.

For advanced information and support, please submit your requests to: support@byteparadigm.com.

2 SPI Operating Mode

2.1 Features

GP Series devices (such as GP-22050) can be configured as a master/analyser device for a *Serial Peripheral Interface (SPI)* bus. The SPI Xpress device offers the same functionalities. The main features of these devices are:

- SPI and 'SPI-like' protocol Master/Analyser for 3- or 4-wire bus architecture
- Controls up to 5 slave devices
- Programmable frequency from 800Hz up to 50MHz
- Programmable polarity for the slave select signals
- Programmable positioning of the slave select signal start and end edges
- Programmable polarity for the write enable signal (3-wire architecture)
- Continuous or non-continuous clock mode
- Programmable level for clock idle state
- Programmable clock edge to generate and capture data
- Programmable latency between write and read access in 3-wire architecture
- Ability to burst the SPI master accesses
- 3 levels of analysis for SPI accesses: oversampled, logical and SPI transfer
- Integrated GTK Wave waveform viewer
- Scripting and logging

2.2 SPI Signals

The standard SPI bus architecture is composed of 4 signals: *SCLK*, *SS*, *MISO* and *MOSI* (refer to Table 1 for detailed description). As the input and output data lines are independent, this architecture can be used to operate in full duplex mode.
A second bus architecture can be implemented using only 3 signals. In this case, a single bidirectional data line is used MISO/MOSI. This bus architecture can then only operate in half-duplex mode.

Table 1: SPI signals description

Signal	Description
SCLK	Serial clock signal generated by the master.
SS	Active low slave select signal. When several slave devise are connected to the same master, SS line is used to activate only one slave for the transfer.
MISO(1)	Master In / Slave Out. Input data line for the master device.
MOSI(1)	Master Out / Slave In. Output data line for the master device.
WE	Write enable signal. Optional signal only used for the 3-wire bus architecture

(1) In the 3-wire architecture, these two signals are combined in a single bidirectional MISO/MOSI signal

The serial clock *SCLK* frequency can be defined from 800 Hz up to 50 MHz. The clock can be generated continuously or not. When the continuous mode is selected, a permanent clock is sent out at the requested frequency. When the non-continuous mode is selected, a *hole* clock is generated. In this case, a clock pulse is only generated when a data bit is shifted in or out of the master device. The idle level of the non-continuous clock is programmable high or low. For example, in non-continuous mode, if 8 bits must be written to the slave, only 8 clock pulses are generated. When no bit is shifted, the clock remains in the idle level programmed by the user.

Up to 5 *SS (Slave Select)* lines can be controlled by the device. These lines are by default active low. They are used to activate a single device when several slaves are connected to the same master. The number of lines that can be driven is programmable from 0 to 5. The number of slave must be set to 0 when a single device is connected to the device and no selection lines must be driven.

Note: When a SS line is driven and when the non-continuous clock mode is enabled, an additional clock pulse is generated by default before to shift the first data bit with the SS signal inactive. Then when the bits are shifted in and/or out, the SS line is driven low (active). At the end of the transfer, an additional clock pulse is generated with the SS line high (inactive). In this mode, if n *bits must be shifted,* n+2 *clock pulses are generated to let the slave detect the rising and falling edge of the SS line. This default addition of clock pulses can be deactivated in C/C++ and TCL modes.*

The *MOSI* line is driven by the master. The SCLK edge used to shift the serialised data out of the master is programmable. The data can be sent out on the rising or falling edge of *SCLK*.
As for the outgoing data line, the edge used to capture the *MISO* input line can be programmed (rising or falling). If *MOSI* is generated on the clock rising edge, then *MISO* is sampled on the clock falling edge and vice versa.
For the 3-wire bus architecture, the *MISO* and *MOSI* lines are combined in a single bidirectional line. When switching between read and write accesses, the data line direction must be inverted in the master and slave devices. Idle state cycles can be introduced between the write and the read accesses to have time to reverse the data line direction. During the idle cycles, the device keeps the data line in high impedance Hi-Z state to avoid conflict on the bus. This latency between write and read can be programmed from 0 to 255.

Notes: 1. The device is cycle accurate and is capable of working without introducing any idle latency cycle between read and write. However this operating mode is not recommended because shorts can occur on the data line due to the AC timing difference between the master and the slave. A minimum value of 1 is recommended. During the write-to-read latency cycles, in non-continuous clock mode, clock pulses can be optionally generated.
2. When working in non-continuous clock mode, no clock pulses is generated during the idle cycles.

A *Write Enable (WE)* signal is generated. The polarity of this signal is programmable. The signal is activated when the first bit of a transfer is shifted out to the selected slave device. It is deactivated when the last bit of the write access has been transferred. During a read access it remains inactive. The main difference between *WE* and *SS*, is that *WE* is only activated when data is written to the slave.

2.3 Signals Mapping on the device connector

Please refer to SPI Xpress / GP Series devices data sheets for signal mapping.

3 SPIC Library

3.1 Functions quick Reference Table

Table 2 gives a list of the functions available in the SPI library. They are grouped by functionality as in the SPIC.h header file

Table 2: Quick reference table of ADWG procedures (by functionality)

Function Prototype	Description
`SPIIC(void)`	Initialises the SPI session using the SPIC library.
`void Terminate(void)`	Closes the SPI sessions. A call to this function is mandatory before closing the application.
`bool IsDeviceReady(void)`	Checks if the device is properly connected to the host PC.
`void SetClockCont(bool Cont)`	Defines the serial clock operating mode: continuous clock or hole clock.
`bool GetClockCont(void)`	Returns the clock mode currently in use.
`void SetMode(int Mode)`	Defines the SPI operating mode.
`int GetMode(void)`	Returns the SPI operating mode.
`void SetWrEn(bool High)`	Sets the WrEn signal active level in 3-wires SPI mode
`bool GetWrEn(void)`	Returns the WrEn signal active level in 3-wires SPI mode
`int SetReqClock(int Freq)`	Defines the requested operating clock frequency (in Hz).
`int GetReqClock(void)`	Returns the current requested operating clock frequency (in Hz).
`int GetSynthClock(void)`	Returns the achieved operating clock frequency (in Hz).
`void SetNrOfSlaves(int NrSlaves)`	Defines the number of SPI slaves connected to the device. A maximum of 5 slaves can be connected to the same device.
`int GetNrOfSlaves(void)`	Returns the number of SPI slaves currently defined.
`void SelectSlave (int SlaveID)`	Defines the ID of the slave to select for the next SPI transfer.
`int GetSelectedSlave(void)`	Returns the ID of the slave currently selected for SPI transfers.
`int SetSSEdges( int SSDelayStart, int SSDelayStop)`	Defines the slave select edges position.
`int GetSSDelayStart(void)`	Returns the slave select start edge positions
`int GetSSDelayStop(void)`	Returns the slave select stop edge positions
`void SetSSActiveLevel(bool Level)`	Sets the SS signal(s) active level.
`bool GetSSActiveLevel(void)`	Return the SS signal(s) active level.
`void SetSSClockMasking(bool Enable)`	Defines the clock masking behaviour with the slave select
`bool GetSSClockMasking(void)`	Returns the clock masking mode with the slave select
`void SetLatencyClockMasking(bool Enable)`	Defines the clock masking behaviour during SPI3 latency
`bool GetLatencyClockMasking(void)`	Returns the clock masking behaviour during SPI3 latency
`void SetBitOrder(bool BitOrder)`	Defines the bit ordering within each data byte.
`bool GetBitOrder(void)`	Returns the bit ordering within data byte.
`Int WriteConfFile( char *FileName, bool Message)`	Writes the current SPI configuration to a file.
`Int ReadConfFile( char *FileName, bool Message)`	Reads the SPI configuration from a file.

Function Prototype	Description
`Int Idle(int NrBits, bool Message = true)`	Runs a given number of Idle cycles.
`Int ShiftWr( int NrBits, void *pDataOut, bool Message)`	Performs a write access to the selected slave.
`int ShiftWrAndRd( int NrBits, char *pDataOut, char *pDataIn, bool Message)`	Performs a simultaneous write and read access to the selected slave (for SPI-4 only)
`int ShiftWrThenRd( int NrBitsOut, int NrBitsIn, char *pDataOut, char *pDataIn, int Latency, bool WREnHigh, bool Message)`	Performs a write access followed by a read access to/from the selected slave. (for SPI-3 only)
`void InitBurst (void)`	Initialises a new burst transfer
`Int ShiftWrBurst( int NrBits, void *pDataOut, bool Message = true)`	Adds a new write access (to the selected slave) to the burst buffer.
`int ShiftWrAndRdBurst( int NrBits, char *pDataOut, char *pDataIn, bool Message)`	Adds a new simultaneous write and read access (to the selected slave) to the burst buffer. This is available in SPI-4 mode only.
`int ShiftWrThenRdBurst( int NrBitsOut, int NrBitsIn, char *pDataOut, char *pDataIn, int Latency, bool WREnHigh, bool Message)`	Adds a write access followed by a read access to/from the selected slave to the burst buffer. This is available in SPI-3 mode only)
`void SendBurst (void)`	Starts the a new burst transfer
`int GetFirstBurstTransfer(void)`	Selects the first SPI transfer to readback the captured data.
`int GetNextBurstTransfer(void)`	Selects the next SPI transfer to readback the captured data.
`int GetBurstData(int nr, char *pData)`	Returns one single data byte from one transfer.
`int GetCapturedData(int NrBytes, void *pData)`	Returns a pointer to the full set of captured data in one transfer.
`int GetScriptLength(char *FileName)`	Returns the loaded script length in number of samples in SPI Master mode.
`int ExecuteScript(char *FileName)`	Executes the specified script in SPI Master mode.
`int GetScriptPos(void)`	Returns the line position in the script being executed.
`int Analyse( unsigned int NrBits, bool SPI4Mode, unsigned int SPI3WrLength, unsigned int SPI3Latency, unsigned int SPI3RdLength, bool Message);`	Starts the SPI Analyser for a given number of samples
`void Abort(void)`	Aborts the running SPI Analyser.

Function Prototype	Description
`int GetCurrPos(void)`	Returns the position within the script being executed (sample index).
`void SetExportFileName(char *FileName)`	Specifies the output file name for the autosave feature in SPI Analyser mode.
`char *GetExportFileName(void)`	Returns the name of output file for the autosave feature in SPI Analyser mode.
`void SetExportFileType(unsigned int FileType)`	Specifies the output file type for the autosave feature in SPI Analyser mode.
`int GetExportFileType(void)`	Returns the selected file type for the autosave feature in SPI Analyser mode.
`void SetAutoSave(bool AutoSave)`	Enables / disables the autosave feature for the SPI Analyser.
`bool GetAutoSave(void)`	Returns the Enable/Disable status of the autosave feature of the SPI Analyser.
`int ExportRawDataFile(char *FileName)`	Exports the analysed data to a file, raw format.
`int ExportRawSPIDataFile(char *FileName)`	Exports the analysed data to a file, raw SPI format (values sampled at the SPI clock edges)
`int ExportDecodedSPIDataFile(char *FileName)`	Exports the analysed data to a file, decoded format.
`int GetLastErr(void)`	Returns the last encountered error code.
`void SetInternalTrigger(bool Internal)`	Specifies the trigger type (internal / external).
`bool GetInternalTrigger(void)`	Returns the current trigger type selection.
`void SetEdgeTrigger(bool Enable)`	Selects edge or level trigger.
`void GetEdgeTrigger(void)`	Returns edge or level trigger.
`void SetCtrlTrigMask(short *pMask)`	Specifies the control trigger mask.
`int GetCtrlTrigMask(void)`	Returns the programmed control trigger mask.
`void SetCtrlTrigPattern(short *pPattern)`	Specifies the trigger pattern.
`int GetCtrlTrigPattern(void)`	Returns the programmed trigger pattern.
`int SetTriggerPos(int Sample)`	Defines the position of the trigger in the run – that is the number of samples before the trigger.
`int GetTriggerPos(void)`	Returns the programmed trigger position.
`int GetOverSampling(void)`	Returns the current clock oversampling.

3.2 Functions details

SPIIC(void)

parameters: ***none***
returns:
description: Initialises an SPI session using the SPIC library. This function must be called at the start of any session using the SPIC library. It enables the control of the device by registering the C session as a client to the *8PI Smart Router*.

void Terminate(void)

parameters: ***none***
returns:
description: Terminates the SPI C session and close the communication with the device. A call to this function is mandatory before closing the application. It is required to unregister the session from the *8PI Smart* Router.

bool IsDeviceReady(void)

parameters: ***none***
returns: Device connection status.

true device ready
false device not ready or not connected

description: Returns the status of the connection with the device. Using this function is not required to be able to communicate with the device. It is provided to check status if needed. When another function requests an access to the device, the communication status is always automatically checked before starting the transfer.

void SetClockCont(bool Cont)

parameters: ***Cont: Boolean parameter defining the clock operating mode***
true continuous clock mode
false hole clock mode

returns:

description: Defines the operating mode for the generated SPI clock. The clock can operate in *continuous* mode where it is permanently applied on the device output pin. In *hole* clock mode, the clock pulses are provided only when data is being transferred (write or read).

bool GetClockCont(void)

parameters: ***none***

returns: Continuous clock status
true continuous clock mode
false hole clock mode

description: Returns the SPI clock mode currently in use.

void SetWrEn (bool High)

parameters: ***High: Boolean.***

returns:

description: Defines the WrEn signal(s) active level (High = 1 for 'high level'; High = 0 for 'low level') (3-wires SPI mode).

bool GetWrEn (void)

parameters: ***none***

returns: 0 or 1. 0 for 'active low'; 1 for 'active high'.

description: Returns the WrEn signal line active level (3-wires SPI mode)

void SetMode(int Mode)

parameters: ***Mode: Defines the SPI operation mode – that is the clock idle level and the edges used to send and capture data bits.***

	Idle Level	*MOSI*	*MISO*
0 (00)	***low***	***falling***	***rising***
1 (01)	***low***	***rising***	***falling***
2 (10)	***high***	***rising***	***falling***
3 (11)	***high***	***falling***	***rising***

returns:

description: Defines the clock idle level and clock edges used to sample or generate data bits. When the clock mode is set to non-continuous (*hole* mode), the level of the clock when no transfer is performed can be programmed to remain high (modes 2/3) or low (modes 0/1).
The SPI clock edges used to generate the data out of the master device (MOSI) or to capture data received by the master (MISO) can also be programmed. According to the selected mode, the data are generated and captured on the rising or falling edge of the SPI clock.

int GetMode(void)

parameters: ***none***

returns: Integer value describing the selected mode.

	Idle Level	*MOSI*	*MISO*
0 (00)	low	falling	rising
1 (01)	low	rising	falling
2 (10)	high	rising	falling
3 (11)	high	falling	rising

description: Returns the SPI synchronisation mode currently in use.

int SetReqClock(int Freq)

parameters: ***Freq: 32-bits integer value providing the requested frequency to be used to generate the SPI clock signal (in Hz). The value can be set between 800 Hz and 50000000 Hz (50MHz).***

800 ≤ Freq ≤ 50000000

The default value is set to 1MHz.

returns: An integer error code; <0 if operation failed

description: Defines the requested frequency for the SPI clock. The parameter of this function is only the requested frequency and not the real frequency that will be generated. The output clock frequency of the device can be programmed with a resolution of 4ns (integer division of a 200MHz reference clock). This means that not all frequencies can be exactly generated. The device always generates the closest frequency corresponding to an integer division of 200MHz and immediately below the requested frequency.
Example:

requested frequency = 124000 Hz
achieved frequency = 123992 Hz

int GetReqClock(void)

parameters: ***none***

returns: Integer value representing the currently requested clock frequency (in Hz)

description: Returns the requested SPI clock frequency expressed in hertz (Hz).

int GetSynthClock(void)

parameters:	***none***
returns:	Integer value between 800 and 50000000 (Hz).
description:	Returns the real output clock frequency generated by the device. This frequency is the closest frequency corresponding to an integer division of 200MHz and immediately below the requested frequency.

void SetNrOfSlaves(int NrSlaves)

parameters:	***NrSlaves: Integer value defining the number of slaves connected to the device. This value can be defined between 0 and 6. The default value is 1.***
returns:	
description:	The device can control up to 5 slaves. This function defines the number of slaves connected to the master device. It configures the number of *Slave Select (SS)* lines that must be used to activate the different slave devices. If the value is set to 0, a single slave device can be connected to the device and no *SS* line is driven.

int GetNrOfSlaves(void)

parameters:	***none***
returns:	Integer value
description:	Returns the current value programmed defining the number of slave devices connected to the device.

void SelectSlave (int SlaveID)

parameters:	***SlaveID: Integer value between 1 and 5. By default, ID=1 is defined.***
returns:	
description:	Defines the ID of the slave that must be selected for the following data transfers.

int GetSelectedSlave (void)

parameters:	***none***
returns:	Integer value between 1 and 5.
description:	Returns the ID of the slave currently selected for the SPI accesses.

int SetSSEdges (int SSDelayStart, int SSDelayStop)

parameters:	***SSDelayStart: slave select start edge delay*** ***SSDelayStop: slave select stop edge delay***
returns:	An integer error code <0 if operation failed
description:	Defines the delays for the edges of the slave select signal. The rising and falling edges can separately be shifted from 0 to 2 quarters of clock period before and after the conventional start end end edge of the transaction. Shifting the SS edges automatically sets up the proper clock oversampling. Valid values for the parameters are ranging from -2 to 1.

int GetSSDelayStart (void)

parameters:	***none***
returns:	*Integer value between -2 and 1.*
description:	Returns the slave select start edge position as defined with the function SetSSEdges

void GetSSDelayStop (void)

parameters: ***none***
returns: *Integer value between -2 and 1.*
description: Returns the slave select stop edge position as defined with the function SetSSEdges

void SetSSActiveLevel (bool Level)

parameters: ***Level: Boolean.***
returns:
description: Defines the SS signal(s) active level (high or low).

bool GetSSActiveLevel (void)

parameters: ***none***
returns: 0 or 1. 0 for 'active low'; 1 for 'active high'.
description: Returns the SS signal(s) line active level.

void SetSSClockMasking (bool Enable)

parameters: ***Enable: Boolean – 1 enables the masking of the clock while the slave select is inactive, 0 disables this feature***
returns:
description: Defines the clock masking behaviour with the slave select.

bool GetSSClockMasking (void)

parameters: ***none***
returns: 0 or 1. 1 for clock masking with slave select; 0 otherwise.
description: Returns the clock masking mode with the slave select.

void SetLatencyClockMasking (bool Enable)

parameters: ***Enable: Boolean – 1 enables the masking of the clock during the SPI3 latency, i.e. while the SPI3 master switched from write to read; 0 disables this feature***
returns:
description: Defines the clock masking behaviour during SPI3 latency.

bool GetLatencyClockMasking (void)

parameters: ***none***
returns: 0 or 1. 1 for clock masking during SPI3 latency; 0 otherwise.
description: Returns the clock masking behaviour during SPI3 latency.

void SetBitOrder (bool BitOrder)

parameters: ***BitOrder: sets MS bit / LS bit first***
returns:
description: Defines the bit ordering within each data byte
0 LS bit first: each byte is sent/read from LSb to MSb
1 MS bit first: each byte is sent/read from MSb down to LSb
Note that the least significant byte of the buffer is always sent first.

bool GetBitOrder (void)

parameters: ***none***
returns: *Boolean*
description: Defines the bit ordering within the data bytes.
0 LS bit first
1 MS bit first

Note that the least significant byte of the buffer is always sent first.

int WriteConfFile (char *FileName, bool Message)

parameters:	****FileName : Output file name, including path.*** ***Message : Enables/disables pop-up messages.***
returns:	*Integer :* Error code: ≥0 if successful; another value if failed.
description:	Writes the current SPI configuration to a file. Refer to [2] for a description of the configuration file format.

int ReadConfFile (char *FileName, bool Message)

parameters:	****FileName : Input file name, including path.*** ***Message : Enables/disables pop-up messages.***
returns:	*Integer:* Error code: ≥0 if successful; another value if failed.
description:	Reads a configuration from a file. Refer to [2] for a description of the configuration file format.

int Idle (int NrBits, bool Message)

parameters:	***NrBits : Number of bits (or SPI clock cycles) to run.*** ***Message : Enables/disables pop-up messages.***
returns:	*Integer:* Error code: ≥0 if successful; another value if failed.
description:	(SPI Master) Holds the SPI interface lines in their default level during a given number of SPI clock cycles.

int ShiftWr(int NrBits, void *pDataOut, bool Message = true)

parameters:	***NrBits:***	***Integer value defining the number of bits to send out to the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***
	pDataOut:	***Pointer to a buffer containing the data bits that must be sent out.***
	Message:	***Boolean flag controlling the generation of dialog box to report error messages.*** ***true dialog box enabled*** ***false dialog box disabled***
	When dialog box are disabled, the error is only reported using the return code.	
returns:	An integer error code. ≥0 if successful.	
description:	Initiates a single write access to the slave device selected by the function *SetSlaveNr()*. The number of bits to transfer is defined using the first parameter *NrBits*. Up to 1.0485.575 bits can be transferred in one burst (1Mb). The second parameter is a pointer to a buffer containing the data bytes to written to the device. This function can be used for both SPI 3- and 4-wire configurations.	

int ShiftWrAndRd(int NrBits, void *pDataOut, void *pDataIn, bool Message = true)

parameters:	***NrBits:***	***Integer value representing the total number of bits to be transferred to/from the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***
	pDataOut:	***Pointer to a buffer containing the data bits that must be sent out.***
	pDataIn:	***Pointer to a buffer that will receives the data bits captured during the read access from the selected slave device.***
	Message:	***Boolean flag controlling the generation of dialog box to report error messages.*** ***true dialog box enabled*** ***false dialog box disabled***
	When dialog box are disabled, the error is only reported using the return code.	
returns:	An integer error code. ≥0 if successful.	
description:	Performs a simultaneous write and read access to the slave device. This function can only be used for a 4-wire SPI configuration. Each time a bit is sent out, a bit is captured. If more bits must be read than written, then the output data buffer must be padded with 0 to contain the correct number of bits.	

int ShiftWrThenRd(int NrBitsOut, int NrBitsIn, void *pDataOut, void *pDataIn, int Latency= 1, bool WREnHigh = true, bool Message = true)

parameters:

NrBitsOut:	***Integer value representing the number of bits to be written to the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***
NrBitsIn:	***Integer value representing the number of bits to be read from the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***
pDataOut:	***Pointer to a buffer containing the data bits that must be sent out.***
pDataIn:	***Pointer to a buffer that will receives the data bits captured during the read access from the selected slave device.***
Latency:	***Integer value defining the number of clock cycles to insert between the write and read access. The value must be defined between 0 and 255.***
WrEnHigh:	***Boolean flag defining the polarity of the write enable signal.*** ***true active high write enable*** ***false active low write enable***
Message:	***Boolean flag controlling the generation of dialog box to report error messages.*** ***true dialog box enabled*** ***false dialog box disabled***

When dialog box are disabled, the error is only reported using the return code.

returns: An integer error code. ≥0 if successful.

description: Performs a write access followed by a read access to the slave device. This function can only be used for a 3-wire SPI configuration. A first write access is performed to the selected slave device. The length of the access is defined by *NrBitsOut*. Then an idle period of programmable length (*Latency*) is waited before starting the read access. This idle period is used to give time to reverse the data signal direction. The latency can be programmed to 0, but it is recommended to program it at least to 1 to avoid shorts/conflict on the data line due to the time needed by the different devices to reverse the direction.
When the idle period is completed, a read access is started. *NrBitsIn* bits are captured.

void InitBurst(void)

parameters: ***none***

returns:

description: Initialises the system for a new SPI burst transfer. Data read during a previous burst and still available in memory is discarded.

int IdleBurst (int NrBits, bool Message)

parameters: ***NrBits : Number of bits (or SPI clock cycles) to run.***
Message : Enables/disables pop-up messages.

returns: *Integer:* Error code: ≥0 if successful; another value if failed.

description: (SPI Master) Holds the SPI interface lines in their default level during a given number of SPI clock cycles. IdleBurst adds a pause *to the burst buffer*. The burst transfer is executed in 2 steps. First, the 'burst' commands are stored in memory; second, they are all executed by calling the SendBurst function.

int ShiftWrBurst(int NrBits, void *pDataOut, bool Message = true)

parameters:	***NrBits:***	***Integer value defining the number of bits to send out to the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***
	pDataOut:	***Pointer to a buffer containing the data bits that must be sent out.***
	Message:	***Boolean flag controlling the generation of dialog box to report error messages.*** ***true dialog box enabled*** ***false dialog box disabled***
	When dialog box are disabled, the error is only reported using the return code.	
returns:	An integer error code. ≥0 if successful.	
description:	Adds a new write access (to the slave device selected by the function *SetSlaveNr()) to the burst buffer*. The burst transfer is started by calling the SendBurst function. The number of bits to transfer is defined using the first parameter *NrBits*. Up to 1.0485.575 bits can be transferred in one burst (1Mb). The second parameter is a pointer to a buffer containing the data bytes to written to the device. This function can be used for both SPI 3- and 4-wire configurations.	

int ShiftWrAndRdBurst(int NrBits, void *pDataOut, void *pDataIn, bool Message = true)

parameters:	***NrBits:***	***Integer value representing the total number of bits to be transferred to/from the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***
	pDataOut:	***Pointer to a buffer containing the data bits that must be sent out.***
	pDataIn:	***Pointer to a buffer that will receives the data bits captured during the read access from the selected slave device.***
	Message:	***Boolean flag controlling the generation of dialog box to report error messages.*** ***true dialog box enabled*** ***false dialog box disabled***
	When dialog box are disabled, the error is only reported using the return code.	
returns:	An integer error code. ≥0 if successful.	
description:	Adds a new simultaneous write and read access (to the slave device) to the burst buffer. The burst transfer is started by calling the SendBurst function. This function can only be used for a 4-wire SPI configuration. Each time a bit is sent out, a bit is captured. If more bits must be read than written, then the output data buffer must be padded with 0 to contain the correct number of bits.	

int ShiftWrThenRdBurst(int NrBitsOut, int NrBitsIn, void *pDataOut, void *pDataIn, int Latency = 1, bool WREnHigh = true, bool Message = true)

parameters: ***NrBitsOut:*** ***Integer value representing the number of bits to be written to the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***

NrBitsIn: ***Integer value representing the number of bits to be read from the slave device. This value must be defined between 1 and 1048575 (1 Mbit).***

pDataOut: ***Pointer to a buffer containing the data bits that must be sent out.***

pDataIn: ***Pointer to a buffer that will receives the data bits captured during the read access from the selected slave device.***

Latency: ***Integer value defining the number of clock cycles to insert between the write and read access. The value must be defined between 0 and 255.***

WrEnHigh: ***Boolean flag defining the polarity of the write enable signal.***
true active high write enable
false active low write enable

Message: ***Boolean flag controlling the generation of dialog box to report error messages.***
true dialog box enabled
false dialog box disabled

When dialog box are disabled, the error is only reported using the return code.

returns: An integer error code. ≥0 if successful.

description: Adds a new write access followed by a read access (to the slave device) to the burst buffer. The burst transfer is started by calling the SendBurst function. This function can only be used for a 3-wire SPI configuration. A first write access is performed to the selected slave device. The length of the access is defined by *NrBitsOut*. Then an idle period of programmable length (*Latency*) is waited before starting the read access. This idle period is used to give time to reverse the data signal direction. The latency can be programmed to 0, but it is recommended to program it at least to 1 to avoid shorts/conflict on the data line due to the time needed by the different devices to reverse the direction.
When the idle period is completed, a read access is started. *NrBitsIn* bits are captured.

void SendBurst(void)

parameters: ***none***

returns:

description: Executes the SPI transfers defined by successive calls to the ShiftWrBurst, ShiftWrAndRdBurst and ShiftWrThenRdBurst function.

int GetFirstBurstTransfer(void)

parameters: ***none***

returns: An integer error code: < 0 if operation failed; ≥ if successful.

description: After the execution of a burst transfer, this command requests the selection of the first SPI transfer to prepare the readback of the captured data. To actually read the readback data, please refer to GetBurstData and GetCapturedData; to select the following burst transfer, please refer to the GetNextBurstTransfer function.

int GetNextBurstTransfer(void)

parameters:	***none***
returns:	An integer error code: < 0 if operation failed; ≥ if successful.
description:	After the execution of a burst transfer, this command requests the selection of the next SPI transfer to prepare the readback of the captured data. To actually read the readback data, please refer to GetBurstData and GetCapturedData; to select the first burst transfer, please refer to the GetFirstBurstTransfer function.

int GetBurstData(int nr, char *pData)

parameters:	***nr : integer used to select the byte index within the selected transfer. For instance, if one single transfer is to return 5 bytes of data, nr can take values from 0 to 4.*** ****pData : pointer to a memory space where the requested data can be stored.***
returns:	An integer error code: < 0 if operation failed; ≥ if successful.
description:	Returns one single data byte from the transfer selected with GetFirstBurstTransfer or GetNextBurstTransfer commands. Please also refer to GetCapturedData

int GetCapturedData(int NrBytes, void *pData)

parameters:	***NrBytes : integer used to specify the total number of bytes to collect from the selected transfer.*** ****pData : pointer to a memory space where the requested data can be stored.***
returns:	An integer error code: < 0 if operation failed; ≥ if successful.
description:	Returns the specified number of bytes from the transfer selected with GetFirstBurstTransfer or GetNextBurstTransfer commands.

int GetScriptLength (char * FileName)

parameters:	***FileName : String – path and file name of the script.***
returns:	Integer representing the script length.
description:	This function analyses the script file given as input parameter and returns its length in number of lines.

int ExecuteScript (char *FileName)

parameters:	***FileName : String – path and file name of the script.***
returns:	*Integer :* ≥0 if execution successful; other value if execution failed.
description:	Starts the execution of the script given as input parameter in SPI Master mode. The transfers specified in the script file is execute in a burst.

int GetScriptPos (void)

parameters:	***none***
returns:	Integer representing the position in the script file.
description:	This function gives the command in the script file that is currently being handled.

int Analyse (unsigned int NrBits,
bool SPI4Mode,
unsigned int SPI3WrLength,
unsigned int SPI3Latency,
unsigned int SPI3RdLength,
bool Message)

parameters:	***NrBits:***	***unsigned integer specifying the number of samples to analyse.***
	SPI4Mode:	***boolean defining the SPI interface type (SPI4 or SPI3)***
	SPI3WrLength:	***in SPI3 mode, length of the write phase in clock cycles***
	SPI3Latency:	***in SPI3 mode, length of the write-to-read latency in clock cycles.***
	SPI3RdLength:	***in SPI3 mode, length of the read phase in clock cycles.***
	Message:	***boolean value enabling / disabling the function pop-up messages; 1 to enable; 0 to disable.***
returns:	*Integer :* ≥0 if analysis successful; other value if execution failed.	
description:	Starts the sampling and the analysis of the specified number of samples from a SPI interface. Note that the analysis is done by oversampling the SPI interface. Hence, the "NrBits" parameters specifies a number of samples taken and not the number of SPI bits sampled. Example: assume one wants so sample a 1MHz SPI bus during 1ms with on oversampling of 10, this would mean "NrBits" must be equal to 10000.	

void Abort (void)

parameters:	***none.***
returns:	
description:	Aborts the execution of a run / script. Requires a multi-threaded environment.

int GetCurrPos (void)

parameters:	***none.***
returns:	Integer, representing the position within the current SPI Master run.
description:	When a script or a shift execution is interrupted with the Abort() command, this function returns the last run sample number reference where the execution stopped.

void SetExportFileName (CString FileName)

parameters:	***FileName: String – path and file name of the export file.***
returns:	
description:	Specifies the name of the export file used with the autosave feature.

char *GetExportFileName (void)

parameters:	***none***
returns:	String – path and file name of the export file.
description:	Returns the name of the export file used with the autosave feature.

void SetExportFileType (unsigned int FileType)

parameters:	***FileType: unsigned integer representing the output file type:*** ***0 : Raw data file***

1 : SPI raw data file
2 : Decoded data file

returns:
description: Specifies the type of the export file used with the autosave feature.
The SPI Analyser proceeds by oversampling the data from the SPI port. It can present the data in 3 formats:
0 : Raw data file: all the samples are given out.
1 : SPI raw data file : the analyser the SPI port signals sampled at the chosen SPI clock edge.
2 : Decoded data file : the SPI transaction are extracted from the bitstream (refer to [2] for a description of the corresponding syntax).

int GetExportFileType (void)

parameters: ***none.***
returns: Integer representing the file type*:*
0 : Raw data file
1 : SPI raw data file
2 : Decoded data file
description: Returns the file type programmed with SetExportFileType() function for the SPI Analyser autosave feature.

void SetAutoSave (bool AutoSave)

parameters: ***AutoSave: boolean – 1 to enable; 0 to disable.***
returns:
description: Enables / disables the SPI Analyser autosave*.*

bool GetAutoSave (void)

parameters: ***none.***
returns: A boolean value.
description: Returns the enable / disable status of the SPI Analyser autosave feature: 1 if enabled; 0 if disabled.

int ExportRawDataFile (char *FileName)

parameters: ***FileName : output export path and file name.***
returns: An integer error code: ≥0 if export successful; other value if export failed.
description: Exports the SPI analysed data to an output file, using the 'raw data' format (please refer to [2] for a description of the SPI Analyser formats).

int ExportRawSPIDataFile (char *FileName)

parameters: ***FileName : output export path and file name.***
returns: An integer error code: ≥0 if export successful; other value if export failed.
description: Exports the SPI analysed data to an output file, using the 'raw SPI data' format (please refer to [2] for a description of the SPI Analyser formats).

int ExportDecodedSPIDataFile (char *FileName)

parameters: ***FileName : output export path and file name.***
returns: An integer error code: ≥0 if export successful; other value if export failed.
description: Exports the SPI analysed data to an output file, using the 'decoded SPI data' format (please refer to [2] for a description of the SPI Analyser formats).

int GetLastErr (void)

parameters:	***none.***
returns:	An integer value.
description:	Returns the last error code returned by the device.

void SetInternalTrigger (bool Internal)

parameters:	***Internal : boolean selecting the trigger type – 1 for internal trigger; 0 for external trigger.***
returns:	
description:	To trigger the SPI Analyser, 2 types of trigger can be selected: - an internal trigger – the SPI Analyser starts immediately when the user runs the Analyse() command; an external trigger, defined onto the device 6 control lines.

bool GetInternalTrigger (void)

parameters:	***none.***
returns:	A boolean – 1 for internal trigger; 0 for external trigger.
description:	Returns the trigger type previously programmed with the SetInternalTrigger() function.

void SetEdgeTrigger (bool Enable)

parameters:	***Enable : boolean selecting edge or level trigger – 1 for edge trigger; 0 for level trigger.***
returns:	
description:	Selects edge or level trigger.

bool GetEdgeTrigger (void)

parameters:	***none.***
returns:	A boolean – 1 for edge trigger; 0 for level trigger.
description:	Returns edge or level trigger.

void SetCtrlTrigMask (short *pMask)

parameters:	****pMask: pointer to a short value. This range of the mask depends on the operating mode:*** • ***Analyser mode : value represents a 6 bit mask and ranges from 0x01 to 0x3F*** • ***Master mode : value represents a 4 bit mask and ranges from 0x02 to 0x1E; bit 0 is used for the write enable signal and bit 5 is used for the clock.***
returns:	
description:	When the external triggering mode is used, the trigger mask selects the control lines to be used as trigger inputs. When a mask bit is set to 0, the corresponding control line is masked for triggering. The mask is given as a pointer to a short value equivalent to the binary value of the mask (example: mask = (Binary)011000 ➔ *pMask points to a short = 24).

void GetCtrlTrigMask (short *pCtrlTrigMask)

parameters:	****pCtrlTrigMask: pointer to a short value. For the ranges, see function SetCtrlTrigMask.***
returns:	
description:	Returns the mask applied on the control lines to detect the external trigger. The value is returned through the **pCtrlTrigMask* pointer.

void SetCtrlTrigPattern(short *pPattern)

parameters:	****pPattern: pointer to a short value. This range of the pattern depends on the operating mode:*** • ***Analyser mode : value represents a 6 bit mask and ranges from 0x01 to 0x3F.*** • ***Master mode : value represents a 4 bit mask and ranges from 0x02 to 0x1E; bit 0 is used for the write enable signal and bit 5 is used for the clock.***
returns:	
description:	Defines the pattern to detect on the trigger inputs to generate the trigger event. The pattern is given as a pointer to a short value equivalent to the binary value of the pattern (example: pattern = (Binary)011000 ➔ *pPattern points to a short = 24).

void GetCtrlTrigPattern (short *pCtrlTrigPattern)

parameters:	****pCtrlTrigPattern: pointer to a short value. For the ranges, see function SetCtrlTrigPattern.***
returns:	
description:	Returns the pattern applied on the control lines to generate a trigger event and start applying data samples on the device system connector. The value is returned through the **pCtrlTrigPattern* pointer.

int SetTriggerPos (int Sample)

parameters:	***Sample: integer value representing the index of the sample in the run where the trigger should be positioned.***
returns:	An integer error code: 0 is successful; another value if unsuccessful.
description:	Use this function to position the trigger after a given number of samples in the total run. Once the sampling run is over, the corresponding number of samples before the trigger is displayed, together with the rest of the run *after* the trigger. *This function can be used in analysermode only.*

int GetTriggerPos (void)

parameters:	***none.***
returns:	A integer representing the trigger position.
description:	Returns the trigger position previously programmed. The trigger position is defined with the sample index at which it is positionned. *This function can be used in analysermode only.*

int GetOverSampling (void)

parameters:	***none.***
returns:	An integer value equal to 1, 2 or 4.
description:	According to the position of the SPI Master SS edges, the device automatically selects the adequate oversampling of its internal clock with respects to the SPI clock. This function returns the selected oversampling.