XSpin User Manual

Document No. 80-20428 Issue 6

Current Issue :- 6, 26th November 2009

Previous Issues :- 5, 8th July 2009
4, 27th March 2008
3, 11th March 2008
2, 4th March 2008
1, 8th September 2008

If your query is not covered in this Manual, or you require further information, please email Heber Customer Support: support@heber.co.uk

The latest version of this document and other technical information can be found on the Heber website: www.heber.co.uk

Copyright © Heber Ltd. 2009. All rights reserved. This document and the information contained therein is the intellectual property of Heber Ltd. and must not be disclosed to a third party without consent. Copies may be made only if they are in full and unmodified.

The information contained in this document is believed to be accurate and reliable. However, Heber Ltd. assumes no responsibility for its use, and reserves the right to revise the documentation without notice.

Precise specifications may change without prior notice.

All trademarks are acknowledged.

HEBER LIMITED
Belvedere Mill
Chalford Stroud
GL6 8NT
UK

Tel +44 (0) 1453 886000
Fax +44 (0) 1453 885013
Email support@heber.co.uk
Website www.heber.co.uk

Document No. 80-20428 Issue 6

CONTENTS

This page intentionally left blank.

1 INTRODUCTION

This document contains the details required to connect an XSpin spin reel control board and operate it through the supplied API software library.

The XSpin board is a member of the XLine family of USB based gaming peripherals; this particular variant of product has been design to specifically operate spinning reels.

2 PRODUCT OVERVIEW

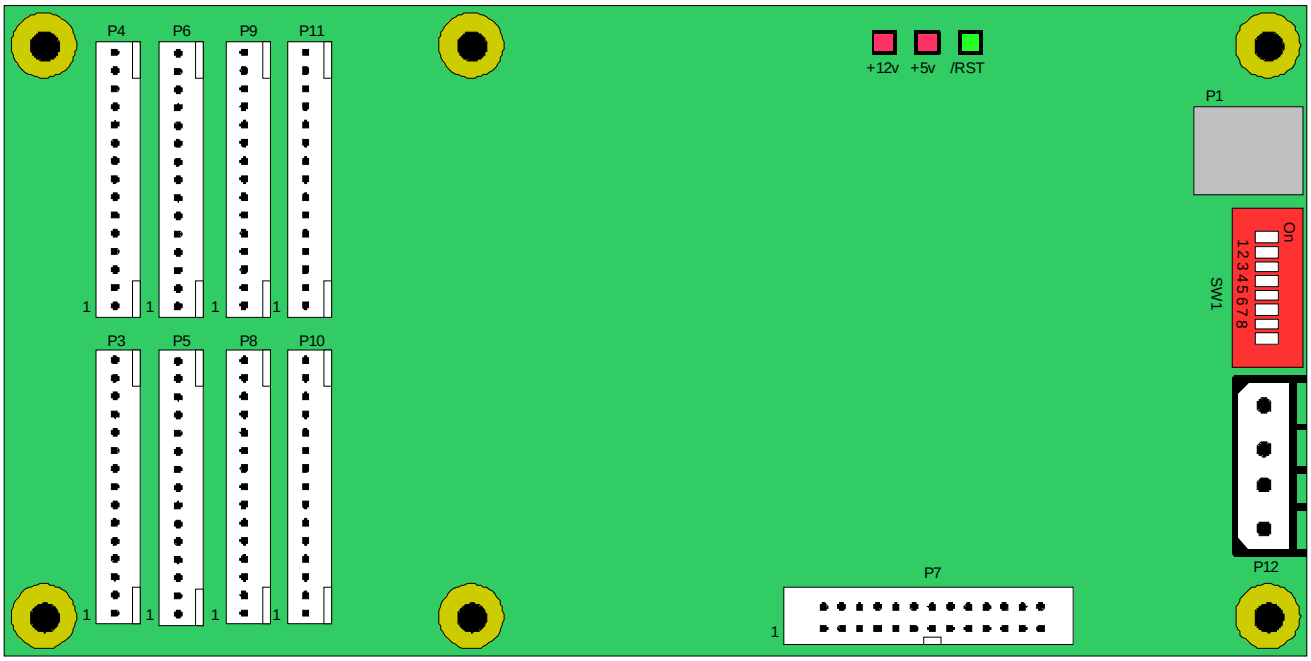
XSpin is able to control, via a USB connection, up to 8 spinning reels simultaneously, with three lamp/LED drive outputs per reel, a further 8 independent digital inputs and 8 independent digital outputs.

The reel outputs can alternatively be used as digital outputs when they are not required to spin reels, giving an extra 7 outputs and one input for each unused reel up to a total of 64 outputs and 16 inputs with no reels connected.

Each reel and its associated lamp/LED are connected independently to the XSpin board. There is also an auxiliary IO connector, USB type B connector and separate power connection.

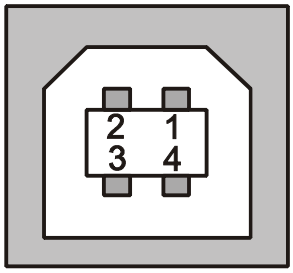
3 HARDWARE DESCRIPTION

The XSpin board as shown below contains 8 reel connectors an auxiliary IO connector, USB and power supply connections.



3.1 USB Connection – P1

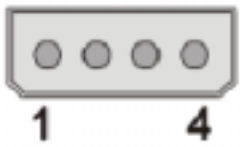
P1 is a USB type B connector for connecting the XSpin board to a PC running Windows or Linux; it operates at USB standard 2.0 but is also USB 1.1 compatible.



1	VUSB
2	Data-
3	Data+
4	GND

3.2 Power Connection – P12

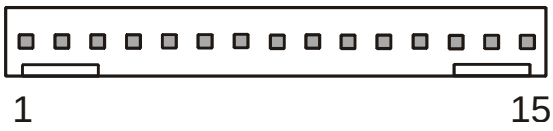
The XSpin board requires an external +5VDC and +12VDC supply to operate fully. The connector P12 is provided for these power supplies. The connection is via a 4 way polarised plug, the same type commonly used to connect to a Hard Disk Drive or CD ROM device.



1	+12v
2	GND
3	GND
4	+5vHDD

3.3 Reel Connections –P3 to P6 and P8 to P10

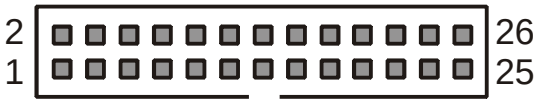
Each reel has its own connection point on the XSpin board which provides power and drive signals to the reel with a return opto signal to the board. There are 8 connectors labelled P3-P6 and P8-P10, where P3 = reel 1, P4 = reel 2 etc.



1	+12v
2	+12v
3	LED / Lamp Output
4	LED / Lamp Output
5	LED / Lamp Output
6	Not Connected
7	+5vHDD
8	Opto input to XSpin
9	+5vHDD
10	GND
11	Reel motor coil 0
12	Reel motor coil 1
13	Reel motor coil 2
14	Reel motor coil 3
15	+12v

3.4 Auxiliary IO Connection – P7

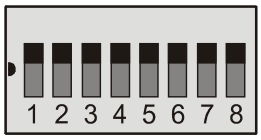
The auxiliary IO connector provides an additional 8 digital inputs and 8 digital outputs for connection to customer applications. Typical applications for the outputs are for connection to additional lamps/LED's; the inputs could be used to monitor switch inputs.



Output 35	1	2	Output 39
Output 43	3	4	Output 47
Output 51	5	6	Output 55
Output 59	7	8	Output 63
GND	9	10	GND
GND	11	12	GND
Input 8	13	14	Input 9
Input 10	15	16	Input 11
Input 12	17	18	Input 13
Input 14	19	20	Input 15
+5vHDD	21	22	+5vHDD
GND	23	24	GND
+12v	25	26	+12v

3.5 DIP Switches -SW1

SW1 provides 8 individual configuration switches; the first three are reserved for the device address and the remaining 5 are user definable. To allow up to a maximum of eight XSpin boards to be connected to a single PC running Windows or Linux simultaneously the first three switches of SW1 are used to define the unique address which is used in software to indentify each board.



3.6 IO Assignment

All of the inputs and outputs on the XSpin board are individually addressable through a number of software library functions. As soon as a reel has been configured, the associated inputs and outputs can no longer be accessed. The table below gives details of which inputs and outputs are assigned to which reel:

Reel	Connector	Motor Outputs	Lamp Outputs	Opto Input
1	P3	OP0...3	OP32...34	IP0
2	P4	OP4...7	OP36...38	IP1
3	P5	OP8...11	OP40...42	IP2
4	P6	OP12...15	OP44...46	IP3
5	P8	OP16...19	OP48...50	IP4
6	P9	OP20...23	OP52...54	IP5
7	P10	OP24...27	OP56...58	IP6
8	P11	OP28...31	OP60...62	IP7

4 API FUNCTIONS

4.1 Function name: XSpin()

Prototype

`XSpin( void )`

Description

This function is the C++ class *constructor* which is called automatically when creating a variable of type XSpin. The function call initialises internal class parameters.

Example

```
XSpin* board;

/* create instance of XSpin class object */
board = new XSpin();
```

4.2 Function name: ~XSpin()

Prototype

`~XSpin( void )`

Description

This function is the C++ class destructor called automatically when destroying a variable of type XSpin. Calling this function performs a clean destruction of the class and its associated internal parameters.

Example

```
XSpin* board;

…

/* destroy instance of XSpin class object */
delete board;
```

4.3 Function name: XSpin(XSpin &)

Prototype

```
XSpin( XSpin & )
```

Description

This function is the C++ class copy constructor. The function provides proper copy construction behaviour and allows temporary XSpin objects to be passed to functions by value or reference.

Example

```
XSpin board;
XSpin bdcopy;

/* initialise board */
if ( FALSE == board.init() )
{
    /* failed to find an XSpin board */
}
else
{
    /* board found and connected */
}

…

/* take a copy of the board object */
bdcopy = board;
```

4.4 Function name: XSpin & operator = (XSpin &)

Prototype

```
XSpin & operator = ( XSpin &)
```

Description

This function is the C++ class assignment constructor which provides proper assignment construction behaviour.

4.5 Function name: Init

Prototype

```
bool Init( void )
```

Description

This function opens a single XSpin USB device. This is a simplified version of the function InitUSBBoard and can be used when there is only one device connected to the system and the address switch setting is ignored.

On exit the function returns success (TRUE) or failure (FALSE), the failure reason can be determined by calling the function `GetLastError()`.

Example

```
XSpin* board;

/* create instance of XSpin class object */
board = new XSpin( );

/* initialise board */
if ( FALSE == board->Init( ) )
{
    /* failed to find a board */
}
else
{
    /* board found and connected*/
}
```

4.6 Function name: InitUSBBoard

Prototype

```
bool InitUSBBoard( unsigned int boardNumber )
```

Description

This function opens an XSpin device. Up to eight XSpin boards can be connected to a single host system dependant on the number of physical connections. Subsequently, all other function calls will be made to this board.

"boardNumber" refers to the address of the board, as determined by the address switch setting on SW1, bits 0-2 (switches 1-3).

- All switches closed (on) corresponds to address 0.
- All switches open (off) corresponds to address 7.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
XSpin* board;

/* create instance of XSpin class object */
board = new XSpin();

/* initialise board number 5 */
if ( FALSE == board->InitUSBBoard( 5 ) )
{
    /* failed to find a board */
}
else
{
    /* board found and connected*/
}
```

4.7 Function Name: Close

Prototype

```
bool Close( void )
```

Description

Closes an XSpin USB device.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
XSpin* board;

/* create instance of XSpin class object */
board = new XSpin();

/* initialise board */
if ( FALSE == board->Init() )
{
    /* failed to find a board */
}
else
{
    /* board found and connected*/
}

…

/* close connection to board */
if ( FALSE == board->Close() )
{
    /* failed to close board connection */
}
else
{
    /* board connected closed */
}
```

4.8 Function Name: GetBoardSpeed

Prototype

```
bool GetBoardSpeed( USBBoardSpeed* pBdSpeed )
```

Description

This function is called to ascertain the USB connection speed at which the currently fitted board is connected, there are three possible values:

- `USBUnknownSpeed` : The board connection speed has not been determined, indicating that the board is not functioning correctly
- `USBFullSpeed` : The board is connected at full speed
- `USBHighSpeed` : The board is connected at high speed

"*pBdSpeed*" points to an enumerated item of type "*USBBoardSpeed*" where the USB connection speed will be stored on successful execution of this function.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
USBBoardSpeed speed;
…

/* determine board connection speed */
if ( TRUE == board->GetBoardSpeed( &speed ) )
{
    /* display connection speed */
    printf( "Connection Speed = %d\n", (int)speed );
}
else
{
    /* error reading connection speed */
}
```

4.9 Function Name: GetProductVersion

Prototype

```
bool GetProductVersion( unsigned char* pVersion )
```

Description

This function reports the XSpin Product version.

"pVersion" points to an *"unsigned char"* where the product version code will be stored on successful execution of this function.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
unsigned char version;
…

/* determine board product version */
if ( TRUE == board->GetProductVersion( &version ))
{
    /* display product version code */
    printf( "Product Version = %d\n", version );
}
else
{
    /* error reading product version code */
}
```

4.10 Function Name: GetDllVersion

Prototype

```
bool GetDllVersion( unsigned char* pVersion )
```

Description

This function reports the version string of the XSpin API. It may be used to check that the correct API has been installed and is being called. It does not require a device handle and can be called without an XSpin board being connected.

"pVersion" points to an array of type *"unsigned char"* where the API version string will be stored on successful execution of this function. The array should be capable of holding at least 10 characters.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
unsigned char version[10];
…

/* determine API version */
if ( TRUE == board->GetDllVersion( &version ) )
{
    /* display dll version string */
    printf( "DLL Version = %s\n", version );
}
else
{
    /* error reading dll version string */
}
```

4.11 Function Name: GetFirmwareVersion

Prototype

```
bool GetFirmwareVersion( unsigned char *version )
```

Description

This function reports the XSpin board firmware version.

"version" points to an *"unsigned char"* where the firmware version shall be stored on successful execution of this function.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
unsigned char version;
…

/* determine board firmware version */
if ( TRUE == board->GetFirmwareVersion( &version ) )
{
    /* display firmware version */
    printf( "Firmware version = %d\n", version );
}
else
{
    /* error reading firmware version */
}
```

4.12 Function Name: GetPowerStatus

Prototype

```
bool GetPowerStatus( XReelPowerStatus* pStatus )
```

Description

This function reports the status of the externally connected 12VDC supply on P12, the two enumerated states are:

- POWER_NOT_CONNECTED
- POWER_CONNECTED

"pStatus" points to an enumerated item of type "*XReelPowerState*" where the power state read from the board will be stored on successful execution of this function.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
XReelPowerStatus status;
…

/* determine board power status */
if ( TRUE == board->GetPowerStatus( &status ))
{
    /* display the power state */
    if ( POWER_NOT_CONNECTED == status )
    {
        printf( "12V Power supply not connected\n" );
    }
    else
    {
        printf( "Power supply connected\n" );
    }
}
else
{
    /* error reading power status */
}
```

4.13 Function Name: GetLastError

Prototype

```
XReelMsg GetLastError( void )
```

Description

This function reports the last error code that has been generated by the board or API in response to a software function call.

Example

```
unsigned char version;
…

/* determine board firmware version */
if ( TRUE == board->GetFirmwareVersion( &version ) )
{
    /* display firmware version */
    printf( "XSpin firmware version = %d\n", version );
}
else
{
    /* error reading firmware version */
    printf( "Error code = %d\n", board->GetLastError() );
}
```

4.14 Function Name: GetInputBit

Prototype

```
bool GetInputBit( XReelInputBit input, bool *pState )
```

Description

This function reports the state of the requested digital input.

"input" is the input channel that is to be read, in the range:

- XREEL_IP0 … XREEL_IP15
- XREEL_SW0 … XREEL_SW7

An error condition shall be generated if the selected input is not a valid input or the input is currently being used by one of the reels.

"pState" points to a "*boolean*" type variable where the state of the requested input will be stored on successful execution of this function. A logic high (1) is represented by the value TRUE and a logic low (0) is represented by the value FALSE.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
bool sw_input;

…

/* read switch setting SW4 */
if ( TRUE == board->GetInputBit( XREEL_SW4, &sw_input ))
{
    /* display the result */
    if ( TRUE == sw_input )
    {
        printf( "Switch SW4 is On\n" );
    }
    else
    {
        printf( "Switch SW4 is Off\n" );
    }
}
else
{
    /* error reading switch input */
}
```

4.15 Function Name: GetInputs

Prototype

```
bool GetInputs( unsigned short* pInputs, unsigned char* pDipSwitch )
```

Description

This function reports the state of the digital inputs and the state of the dip switch.

"pInputs" points to an "*unsigned short*" type variable where the state of the inputs are stored on successful execution of this function. The inputs are stored as a bit pattern where bit 0 refers to IP0 and bit 15 refers to IP15.

"pDipSwitch" points to an "*unsigned char*" type variable where the state of the configuration dip switch will be stored on successful execution of this function. The switch inputs are stored as a bit pattern where bit 0 refers to SW0 and bit 7 refers to SW7.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
unsigned short inputs;
unsigned char switch;
…

/* read inputs */
if ( TRUE == board->GetInputs( &inputs, & switch ) )
{
    /* display the result */
    printf( "Inputs = %04x\nSwitches = %02x\n", inputs, switch );
}
else
{
    /* error reading inputs */
}
```

4.16 Function Name: GetChangedInputs

Prototype

```
bool GetChangedInputs( unsigned short* pInputs )
```

Description

This function reports the digital inputs that have changed state since the inputs were last read.

"*pInputs*" points to an "*unsigned short*" type variable where the state of the changed inputs are stored on successful execution of this function. The changed inputs are stored as a bit pattern where bit 0 refers to IP0 and bit 15 refers to IP15.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
unsigned short inputs;
…

/* read changed inputs */
if ( TRUE == board->GetChangedInputs( &inputs ) )
{
    /* display the result */
    printf( "Inputs Changed = %04x\n", inputs );
}
else
{
    /* error reading inputs */
}
```

4.17 Function Name: SetOutputBit

Prototype

```
bool SetOutputBit( XReelOutputBit output, bool state )
```

Description

This function modifies the output state of a selected output on the XSpin board.

"*output*" is the output channel that is to be set, in the range:

- XREEL_OP0 … XREEL_OP63

An error condition will be generated if the selected output is not a valid output or the output is currently being used by one of the reels.

"*state*" is the logical state to which the selected output will be set, a logic high (1) is represented by the value TRUE and a logic low (0) is represented by the value FALSE.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* set output OP63 on */
if ( FALSE == board->SetOutputBit( XREEL_OP63, TRUE ) )
{
    /* error setting output */
}
```

4.18 Function Name: SetOutputs

Prototype

```
bool SetOutputs(    unsigned int outputStates31_0,
                    unsigned int outputStates63_32 )
```

Description

This function modifies the output state of all of the outputs on the XSpin board, where the output is not assigned to drive a reel.

"*outputStates31_0*" is the state of the outputs OP0…OP31, where bit 0 refers to OP0 and bit 31 refers to OP31. A logic high (1) is represented by the value TRUE and a logic low (0) is represented by the value FALSE. Note that any outputs assigned to drive a reel will not be affected by this function.

"*outputStates63_32*" is the state of the outputs OP0…OP31, where bit 0 refers to OP32 and bit 31 refers to OP63. A logic high (1) is represented by the value TRUE and a logic low (0) is represented by the value FALSE.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
unsigned int out_low;
unsigned int out_high;

…

/* assign outputs to be set */
out_low  = 1 << 24;
out_high = ( 1 << 6 ) | ( 1 << 31 );

/* set outputs OP24, OP38 and OP63 */
if ( FALSE == board->SetOutputs( out_low, out_high ) )
{
    /* error setting outputs */
}
```

4.19 Function Name: ModifyOutputs

Prototype

```
bool ModifyOutputs( unsigned int set31_0,
                    unsigned int set63_32,
                    unsigned int clear31_0,
                    unsigned int clear63_32 )
```

Description

This function modifies the state of individual outputs on the XSpin board, where the output is not assigned to drive a reel.

Setting a bit in the parameter "*set31_0*" will cause the corresponding output OP0…OP31, where bit 0 refers to OP0 and bit 31 refers to OP31, to be set with any remaining bits not set remaining unchanged.

Setting a bit in the parameter "*set63_32*" will cause the corresponding output OP32…OP63, where bit 0 refers to OP32 and bit 31 refers to OP63, to be set with any remaining bits not set remaining unchanged.

Setting a bit in the parameter "*clear31_0*" will cause the corresponding output OP0…OP31, where bit 0 refers to OP0 and bit 31 refers to OP31, to be cleared with any remaining bits not cleared remaining unchanged.

Setting a bit in the parameter "*clear63_32*" will cause the corresponding output OP32…OP63, where bit 0 refers to OP32 and bit 31 refers to OP63, to be cleared with any remaining bits not cleared remaining unchanged.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Considerations

Any outputs assigned to drive a reel will not be affected by this function.

Example

```
unsigned int set_low, set_high;
unsigned int clear_low, clear_high;

…

/* assign outputs to be set or cleared*/
set_low    = 1 << 24;
set_high   = ( 1 << 6 ) | ( 1 << 31 );
clear_low  = 0;
clear_high = ( 1 << 8 ) | ( 1 << 19 );

/* set outputs OP24, OP38 and OP63 – clear OP40 and OP51 */
if ( FALSE == board->ModifyOutputs(  set_low, set_high,
                                     clear_low, clear_high ) )
{
    /* error modifying outputs */
}
```

4.20 Function Name: ConfigureReels

Prototype

```
bool ConfigureReels(     unsigned int numberOfReels,
                         unsigned int halfStepsPerTurn,
                         unsigned int stepsPerSymbol )
```

Description

This function configures the XSpin board prior to spinning reels.

"*numberOfReels*" is the number of reels to be configured, in the range 1…8.

"*halfStepsPerTurn*" is the number of half steps required to turn a reel through 360 degrees.

"*stepsPerSymbol*" is the number of steps that are required to move a reel from one symbol shown on the win line to the next symbol being shown on the win line.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* Four 200 step reels with 40 steps per symbol */
if ( FALSE == board->ConfigureReels( 4, 200, 40 ) )
{
    /* error configuring reels */
}
```

4.21 Function Name: SpinReels

Prototype

```
bool SpinReels(    unsigned char reelNumber,
                   char directionStepSize,
                   unsigned int numberOfSteps )
```

Description

This function starts a previously configured reel spinning in the specified direction, for the specified number of steps. The reel will be stopped automatically on the completion of the specified number of steps.

"*reelNumber*" specifies the reel that is to be spun.

"*directionStepSize*" is the size of a step and the direction in which the reel is to be moved, the options are:

- Half step forward = 1
- Full step forward = 2
- Half step backwards = -1
- Full step backwards = -2

"*numberOfSteps*" is the number of steps that the reel is to be moved.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 2 forward 80 full steps */
if ( FALSE == board->SpinReels( 2, 2, 80 ) )
{
    /* error spinning reel */
}
```

4.22 Function Name: SpinRampUp

Prototype

```
bool SpinRampUp( unsigned char reel, XReelRampTable* rampTable )
```

Description

This command must be issued for reel number <reel>. before spinning a reel for the first time.

This function configures the ramp table for a specific reel; the ramp table is a table of values that specifies the behaviour of the reel during the ramp up phase of the reel from idle to spinning speed.

The variable <rampTable> must be of structure type <XReelRampTable>. This structure is an array of delays, in milliseconds, with the first byte indicating the length of the data table that follows (up to a maximum of 60 data entries). The **first** millisecond delay value is a power up delay and will not cause the reel to step. It is the amount of time that the reel must be on full power before stepping can start.

Each delay entry in the table has a maximum value of 255ms.

For example, a ramp up table of {6,200,50,40,30,20,18} will specify a power-up delay of 200ms followed by a ramp of five steps of 50ms, 40ms, 30ms, 20ms and 18ms. It also defines the stepping rate to be used for the remainder of the spin until the reel is ready to ramp down: it will use the last entry in the table, which in this case is 18ms.

"*reel*" is the reel that is to be configured.

"*rampTable*" points to a structure of type "*XReelRampTable*" containing the ramp table for the specified reel.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* create and configure table */
XReelRampTable table = { 5, { 100, 57, 30, 25, 21 } };

…

/* configure the ramp up table for reel 3 */
if ( FALSE == board->SpinRampUp( 3, &table ) )
{
    /* error setting ramp table */
}
```

4.23 Function Name: SpinRampDown

Prototype

```
bool SpinRampDown( unsigned char reel, XReelRampTable* rampTable )
```

Description

This command must be issued for reel number <reel>. before spinning a reel for the first time.

This function configures the ramp table for a specific reel; the ramp table is a table of values that specifies the behaviour of the reel during the ramp down phase of the reel from spinning speed to idle.

The variable <rampTable> must be of structure type <XReelRampTable>. This structure is an array of delays, in milliseconds, with the first byte indicating the length of the data table that follows (up to a maximum of 60 data entries). The **last** millisecond delay value is a power down delay and will not cause the reel to step. It is the amount of time that the reel must remain on full power at the end of the ramp before the motor power is dropped back to a lower level by chopping the drive.

Each delay entry in the table has a maximum value of 255ms.

For example, ramp table of {4,19,20,30,250} will specify a ramp of three steps of 19ms, 20ms and 30ms, followed by a period of 250ms at full power, after which the stepper motor power is reduced by chopping the stepper motor drive.

"*reel*" is the reel that is to be configured.

"*rampTable*" points to a structure of type "*XReelRampTable*" containing the ramp table for the specified reel.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* create and configure table */
XReelRampTable table = { 4, { 23,  23, 58, 200 } };

…

/* configure the ramp down table for reel 1 */
if ( FALSE == board->SpinRampDown( 1, &table ) )
{
    /* error setting ramp table */
}
```

4.24 Function Name: SetDutyCycle

Prototype

```
bool SetDutyCycle( unsigned char reelNumber,
                   unsigned char offPeriod,
                   unsigned char onPeriod )
```

Description

This function configures the duty cycle timing of the reels while idle; the maximum duration of on/off time is 255mS. The default is 5mS on and 5mS off.

"*reelNumber*" is the reel that is to be configured.

"*offPeriod*" is the period of time in milliseconds the motor winding of the reel is off.

"*onPeriod*" is the period of time in milliseconds the motor winding of the reel is on.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* set 5ms on, 10ms off duty cycle for reel 2*/
if ( FALSE == board->SetDutyCycle( 2, 10, 5 ) )
{
    /* error setting duty cycle */
}
```

4.25 Function Name: GetReelStatus

Prototype

```
bool GetReelStatus( unsigned char reelNumber, XReelStatus* pStatus )
```

Description

On success this function returns status information about the specified reel.

"*reelNumber*" is the reel from which to read the status.

"*pStatus*" points to a structure of type "*XReelStatus*" where the status information is stored. This structure contains the following information:

- The current reel position (0-65535).
- The last reel position error value (-32768 to +32767), which is the difference between actual position and expected position when the "zero position" tab was last passed.
- The reel busy state, which will be TRUE if busy and the last spin request has not yet been completed.
- The tampered state. When FALSE, no tampering has occurred. When TRUE, the reel has been moved, and a tab detected, when the reel should be idle. The tampered state can be cleared by calling ClearTamperedState().
- The synchronisation state. When TRUE, the position counter will be copied into the error counter as the "zero position" tab is passed and the synchronisation state will remain set to TRUE. It can be set to FALSE again by ReelSynchroniseEnable().
- The number of tab pulses detected during the last spin.

The synchronisation state is initially set to FALSE. In order to synchronise, it is necessary to perform a full 1.5 revolution spin. The XSpin board supports multiple tabs running around the reel circumference, however the "zero position" tab must always be the thinnest. Once the initial spin is complete, and at least one tab was detected, the synchronisation state will be set TRUE.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
XReelStatus status;
…

/* determine status of reel 4 */
if ( FALSE == board->GetReelStatus( 4, &status ) )
{
    /* error reading status */
}
```

4.26 Function Name: ClearTamperedState

Prototype

```
bool ClearTamperedState( unsigned char reelNumber )
```

Description

This function clears the tampered state for the specified reel.

"*reelNumber*" is the reel to clear.

The tampered state is automatically set TRUE when the reel is moved (and a tab is detected) when the reel should be idle. The tampered state can be obtained by calling GetReelStatus().

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* clear tampered state for reel 2 */
if ( FALSE == board->ClearTamperedState( 2 ) )
{
    /* error detected */
}
```

4.27 Function Name: ReelSynchroniseEnable

Prototype

```
bool ReelSynchroniseEnable( unsigned char reelNumber )
```

Description

This function initiates the reel synchronisation process for the specified reel.

"*reelNumber*" is the reel that requires synchronisation.

After power-up or a call to ReelSynchroniseEnable(), the synchronisation state will be FALSE. The reels must be configured using ConfigureReels(), then ramp tables must be defined for each reel using SpinRampUp() and SpinRampDown() functions. Finally, each reel is spun through at least one turn using SpinReels(). A call to GetReelStatus() should show that each reel is now synchronised with zero error count.

Synchronisation state: When FALSE, the reel position counter will be set to 0 as the opto-detector is passed, and the synchronisation state will be set to TRUE. When TRUE, the position counter will be copied into the error counter as the opto-detector is passed and the synchronisation state will remain set to TRUE. It can only be set to FALSE again by ReelSynchroniseEnable().

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* synchronise reel 2 */
if ( FALSE == board->ReelSynchroniseEnable( 2 ) )
{
    /* error detected */
}
```

4.28 Function Name: SpinReelStepForward

Prototype

```
bool SpinReelStepForward(      unsigned char reelNumber,
                               unsigned char stepSize,
                               unsigned int numberOfSteps )
```

Description

This function starts a previously configured reel spinning in a forward direction, for the specified number of steps. The reel will be stopped automatically on the completion of the specified number of steps.

"*reelNumber*" specifies the reel that is to be spun.

"*stepSize*" is the size of a step which the reel is to be moved, the options are:

- Half step forward = 1
- Full step forward = 2

"*numberOfSteps*" is the number of steps that the reel is to be moved.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 2 forward 80 full steps */
if ( FALSE == board-> SpinReelStepForward( 2, 2, 80 ) )
{
    /* error spinning reel */
}
```

4.29 Function Name: SpinReelStepBack

Prototype

```
bool SpinReelStepBack(    unsigned char reelNumber,
                          unsigned char stepSize,
                          unsigned int numberOfSteps )
```

Description

This function starts a previously configured reel spinning in a reverse direction, for the specified number of steps. The reel will be stopped automatically on the completion of the specified number of steps.

"*reelNumber*" specifies the reel that is to be spun.

"*stepSize*" is the size of a step which the reel is to be moved, the options are:

- Half step backwards = 1
- Full step backwards = 2

"*numberOfSteps*" is the number of steps that the reel is to be moved.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 1 backward 40 full steps */
if ( FALSE == board-> SpinReelStepBack( 1, 2, 40 ) )
{
    /* error spinning reel */
}
```

4.30 Function Name: SpinReelSymbolForward

Prototype

```
bool SpinReelSymbolForward(     unsigned char reelNumber,
                                unsigned char stepSize,
                                unsigned char numberOfSymbols )
```

Description

This function starts a previously configured reel spinning in a forward direction, for the specified number of reel symbols. The reel will be stopped automatically on the completion of the specified number of steps.

"*reelNumber*" specifies the reel that is to be spun.

"*stepSize*" is the size of a step which the reel is to be moved. The options are:

- Half step forward = 1
- Full step forward = 2

"*numberOfSymbols*" is the number of reel symbols that the reel is to be moved.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
…
/* spin reel 2 forward 8 symbols */
if ( FALSE == board->SpinReelSymbolForward( 2, 2, 8 ) )
{
    /* error spinning reel */
}
```

4.31 Function Name: SpinReelSymbolBack

Prototype

```
bool SpinReelSymbolBack( unsigned char reelNumber,
                         unsigned char stepSize,
                         unsigned int numberOfSymbols )
```

Description

This function starts a previously configured reel spinning in a reverse direction, for the specified number of reel symbols. The reel will be stopped automatically on the completion of the specified number of steps.

"*reelNumber*" is the reel that is to be spun.

"*stepSize*" is the size of a step which the reel is to be moved. The options are:

- Half step backwards = 1
- Full step backwards = 2

"*numberOfSymbols*" is the number of reel symbols that the reel is to be moved.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 1 backward 12 symbols */
if ( FALSE == board-> SpinReelSymbolBack( 1, 2, 12 ) )
{
    /* error spinning reel */
}
```

4.32 Function Name: SetReelLamps

Prototype

```
bool SetReelLamps( unsigned char reelNumber, bool state )
```

Description

This function controls the state of the three LED/Lamp outputs that may be attached to a specified reel.

"*reelNumber*" is the reel that the lamps are attached to.

"*state*" is the state that the lamps are to be switched to.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* flash the lamps on reel 4 */
int count = 5;

while ( count-- )
{
    if ( FALSE == board-> SetReelLamps( 4, TRUE ) )
    {
        /* error controlling lamps */
    }

    Sleep( 500 );

    if ( FALSE == board-> SetReelLamps( 4, FALSE ) )
    {
        /* error controlling lamps */
    }

    Sleep( 500 );
}
```

4.33 Function Name: SpinStartSkill

Prototype

```
bool SpinStartSkill( unsigned char reel, char directionStepSize )
```

Description

This function starts a previously configured reel spinning in the specified direction, for an undefined number of steps. The reel will only be stopped when the SpinStopSkill() function is called, the stopping position will be aligned to the next nearest symbol providing that the reel was aligned on a symbol prior to calling this feature.

"*reel*" specifies the reel that is to be spun.

"*directionStepSize*" is the size of a step and the direction in which the reel is to be moved, the options are:

- Half step forward = 1
- Full step forward = 2
- Half step backwards = -1
- Full step backwards = -2

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* start the 'skill stop' feature on reel 2 */
/* using full stepping mode */
if ( FALSE == board->SpinStartSkill( 2, 2 ) )
{
    /* error spinning reel */
}
```

4.34 Function Name: SpinStopSkill

Prototype

```
bool SpinStopSkill( unsigned char reelNumber )
```

Description

This function stops a spinning reel, which has previously been started by calling the SpinSkillStart() function. The reel will be stopped in a position where it is aligned to the next nearest symbol providing that the reel was aligned on a symbol prior to starting the reel spinning.

"*reelNumber*" specifies the reel that is to be stopped.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* stop reel 2 running under the 'skill stop' feature */
if ( FALSE == board->SpinStopSkill( 2 ) )
{
    /* error stopping reel */
}
```

4.35 Function Name: StopCurrentSpinAtPosition

Prototype

```
bool StopCurrentSpinAtPosition( unsigned char reelNumber,
                                unsigned int position )
```

Description

This function stops a spinning reel the next time it reaches the required position. This function is valid for any type of spin (including skill-stop). The reel must be synchronised prior to calling this function.

"*reelNumber*" specifies the reel that is to be stopped.

"*position*" specifies the position at which to stop the reel.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* stop reel 2 at position 100 */
if ( FALSE == board->StopCurrentSpinAtPosition( 2, 100 ) )
{
    /* error stopping reel */
}
```

5 ENHANCED REEL SPINNING API

5.1 Introduction

The Enhanced Reel Spinning API can be used as an alternative method of spinning reels. The API provides a suite of "higher level" functions that are symbol aware and that conceal some of the details involved in spinning reels. For example, the duty cycle and the ramp tables are automatically configured based upon the reel type and the requested spin speed. In addition, the API allows you to use a combination of different reel types. The Enhanced API functions rely on a *Reel Definition File* that contains all of the parameters needed to control the reels. All of the Enhanced API function names begin with "Vendor".

5.2 The Reel Definition File

The Reel Definition File is a single file that contains data for all of the supported reel types. The file is in XML format, which is text-based and human readable.

Each reel type has a unique name, for example, "REEL_00003". A sample Reel Definition File is shown in section 5.2.2 below.

5.2.1 Reel Definition File Notes

It is expected that a sample Reel Definition File will be provided by Heber but it may be necessary for customers to edit this file to fine-tune the reel behaviour or to add new ramp tables. Here are some Reel Definition File guidelines:

1. Each reel definition must have exactly one "params" section, exactly one "symbol-table" section and at least one "ramp-table" section.

2. Symbol table “steps per turn” parameter is in full steps.

3. Symbol table position entries are in half steps.

4. Symbol table entries must be unique and listed in increasing numerical order (i.e. monotonic).

5. Tamper detection method. At present there are two methods:
   - 0 – Tamper Detect Disabled.
   - 1 - Tamper Detect Static.

6. Tab Type. At present there are two types:
   - 0 – Narrow Tab.
     This type covers the traditional single tab reel plus reels where there are multiple tabs but only one narrow one.
   - 1 – Narrow Slot.
     This type covers reels where there are multiple slots but only one narrow slot.
   - 2 - Wide Tab.
     This type covers reels where there are multiple tabs but only one wide tab.

7. Position detection method. At present this is sent down to the XSpin board but is not used.

8. Error Deadband. This is a small deadband to eliminate false errors.

5.2.2 Sample Reel Definition File

```
<?xml version="1.0" ?>

<heber-reel-definitions>   <!-- the one and only XML root element -->

 <!-- ************************************************************ -->

 <reel-def type="REEL_00001">

  <params
   position-detection-method="0"
   tamper-detection-method="1"
   duty-cycle-off="1"
   duty-cycle-on="6"
   tab-width = "14"
   tab-type="1"
   tab-offset="8"
   error-deadband="2">
  </params>

  <symbol-table steps-per-turn="200" symbols-per-turn="20"
   s000=  "0" s001= "20" s002= "40" s003= "60" s004= "80" s005="100" s006="120"
   s007="140" s008="160" s009="180" s010="200" s011="220" s012="240" s013="260"
   s014="280" s015="300" s016="320" s017="340" s018="360" s019="380" >
  </symbol-table>

  <ramp-table speed = "24" step-type="full">
   <ramp-up num-entries="7"
      r000="100" r001= "37" r002= "24" r003= "13" r004= "18" r005= "15" r006= "13" >
   </ramp-up>
   <ramp-down num-entries="4"
      r000= "22" r001= "13" r002= "41" r003= "200" >
   </ramp-down>
  </ramp-table>

  <ramp-table speed = "100" step-type="full">
   <ramp-up num-entries="23"
      r000="255" r001= "20" r002= "12" r003= "10" r004=  "7" r005=  "7" r006=  "6"
      r007=  "6" r008=  "5" r009=  "5" r010=  "4" r011=  "5" r012=  "4" r013=  "4"
      r014=  "4" r015=  "3" r016=  "4" r017=  "3" r018=  "4" r019=  "3" r020=  "3"
      r021=  "6" r022=  "3" >
   </ramp-up>
   <ramp-down num-entries="18"
      r000=  "7" r001=  "4" r002=  "3" r003=  "3" r004=  "4" r005=  "4" r006=  "3"
      r007=  "4" r008=  "5" r009=  "5" r010=  "5" r011=  "5" r012=  "6" r013=  "7"
      r014= "10" r015= "14" r016= "20" r017="200" >
   </ramp-down>
  </ramp-table>

 </reel-def>

 <!-- ************************************************************ -->

 <reel-def type="REEL_00002">

  <!--  next reel data here etc...... -->

 </reel-def>

 <!-- ************************************************************ -->

 </heber-reel-definitions>
```

5.3 Function Name: VendorSetReelFilePath

Prototype

```
bool VendorSetReelFilePath( char* pPath )
```

Description

This function sets the name and location of the Reel Definition File. This function should be called before `VendorConfigureReel()`. If this function is not called then the API will attempt to use a file called “heber-reel-definitions.xml” in the current working directory.

"*pPath*" is the fully qualified file path to the Reel Definition File.

Note that, in C, a backslash character in a string must be entered as a double backslash. Alternatively, you can use a single forward slash character. For example, these two C strings represent the same file path:

```
"f:\\reel\\heber-reel-definitions.xml"
"f:/reel/heber-reel-definitions.xml"
```

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* set the file path */
if ( !board->VendorSetReelFilePath("f:\\heber-reel-definitions.xml") )
{
        /* error - file path */
}
```

5.4 Function Name: VendorConfigureReel

Prototype

```
bool VendorConfigureReel( unsigned char reelNumber, char* pReelType )
```

Description

This function tells XSpin what type of reel is being used in this position. The API will attempt to open the Reel Definition File and find the data for the given reel. Each reel can be a different reel type.

This function must be called before any of the following reel functions can be used.

"*reelNumber*" is the number of the reel to configure.

"*pReelType*" is the name of the reel type and must match an entry in the Reel Definition File.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* configure reel 0 */
if ( !board->VendorConfigureReel(0, "REEL_00005") )
{
       /* error – configuring reel 0 */
}

/* configure reel 3 */
if ( !board->VendorConfigureReel(3, "REEL_00004") )
{
       /* error – configuring reel 3 */
}
```

5.5 Function Name: VendorSynchroniseReel

Prototype

```
bool VendorSynchroniseReel( unsigned char reelNumber )
```

Description

This function asks XSpin to synchronise the given reel. The reel will spin forwards to search for the home position and, assuming it is located, this function will stop the reel at the home position. The tamper flag will be cleared.

"*reelNumber*" is the number of the reel to synchronise.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* synchronise reel 2 */
if ( !board->VendorSynchroniseReel(2) )
{
        /* error synchronising reel */
}
```

5.6 Function Name: VendorSpinReelNumberOfSymbols

Prototype

```
bool VendorSpinReelNumberOfSymbols( unsigned char reelNumber,
                                    unsigned int numberOfSymbols,
                                    int spinSpeed )
```

Description

This function will spin one reel forwards or backwards by the required number of symbols at the requested speed. The requested speed will be rounded to the nearest value that is available in the reel definition data and, if the movement distance is relatively short, the speed may be reduced to allow for the motor ramp times. If the requested symbol movement cannot be achieved with any the available ramp tables then the reel will not spin and this function will return "failure".

"*reelNumber*" is the number of the reel to spin.

"*numberOfSymbols*" is the number of symbols to spin. This must always be positive.

"*spinSpeed*" is the spin speed in rpm. A negative speed will cause a backwards spin.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 2 forwards 14 symbols at 100 rpm*/
if ( !board->VendorSpinReelNumberOfSymbols(2, 14, 100 ) )
{
       /* error spinning reel */
}

/* spin reel 2 backwards 5 symbols at 20 rpm*/
if ( !board->VendorSpinReelNumberOfSymbols(2, 5, -20 ) )
{
       /* error spinning reel */
}
```

5.7 Function Name: VendorSpinReelToSymbol

Prototype

```
bool VendorSpinReelToSymbol( unsigned char reelNumber,
                             unsigned int symbolNumber,
                             int spinSpeed )
```

Description

This function will spin forwards or backwards to the target symbol at the requested speed. The requested speed will be rounded to the nearest value that is available in the reel definition data. If the distance to the target symbol is too short for the requested speed then the reel will spin a full revolution then stop at the target symbol.

"*reelNumber*" is the number of the reel to spin.

"*symbolNumber*" is the target symbol.

"*spinSpeed*" is the spin speed in rpm. A negative speed will cause a backwards spin.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 1 backwards to symbol 7 at 100 rpm */
if ( !board->VendorSpinReelToSymbol(1, 7, -100 ) )
{
       /* error spinning reel */
}
```

5.8 Function Name: VendorSpinReelStartSkill

Prototype

```
bool VendorSpinReelStartSkill( unsigned char reelNumber,
                               int spinSpeed )
```

Description

This function starts an indefinite spin forwards or backwards. The requested speed will be rounded to the nearest value that is available in the reel definition data.

"*reelNumber*" is the number of the reel to spin.

"*spinSpeed*" is the spin speed in rpm. A negative speed will cause a backwards spin.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* spin reel 0 forwards at 20 rpm */
if ( !board->VendorSpinReelStartSkill( 0, 20 ) )
{
       /* error spinning reel */
}
```

5.9 Function Name: VendorSpinReelStopSkill

Prototype

```
bool VendorSpinReelStopSkill( unsigned char reelNumber )
```

Description

This function stops an indefinite spin at the next available symbol position, allowing for ramp down needs.

"*reelNumber*" is the number of the reel to stop.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* stop reel 0 */
if ( !board->VendorSpinReelStopSkill(0) )
{
        /* error stopping spin */
}
```

5.10 Function Name: VendorStopReelSpinAtPosition

Prototype

```
bool VendorStopReelSpinAtPosition( unsigned char reelNumber,
                                   unsigned int symbolPosition )
```

Description

This function stops an indefinite spin at the given symbol position, allowing for ramp down needs. If the target symbol is too close then the reel will spin one whole revolution and then stop at the target symbol.

"*reelNumber*" is the number of the reel to configure

"*symbolPosition*" is the target symbol.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function `GetLastError()`.

Example

```
/* stop reel 0 at symbol 13 */
if ( !board->VendorStopReelSpinAtPosition(0, 13) )
{
       /* error stopping reel */
}
```

5.11 Function Name: VendorGetReelStatus

Prototype

```
bool VendorGetReelStatus( unsigned char reelNumber,
                          XReelStatusEx* pStatusEx )
```

Description

This function gets the extended status of a reel. The extended status information will be written to an *XReelStatusEx* structure. This structure contains all of the information in the standard *XReelStatus* structure plus some additional fields relating to the Enhanced API.

"*reelNumber*" is the reel from which to read the status.

"*pStatusEx*" is a pointer to the XReelStatusEx structure that will receive the status.

XReelStatusEx is defined as follows:

```
typedef struct
{
    unsigned short   position;
    unsigned short   error;
    unsigned char    busy;
    unsigned char    synchronised;
    unsigned char    tamperedState;
    unsigned short   pulsesDetected;
    unsigned int     symbolPosition;
    signed char      symbolError;
    signed int       currentSpeed;
} XReelStatusEx;
```

The first 6 fields are identical to those in the XReelStatus structure. The new fields are as follows:

symbolPosition is the closest symbol to the current position.

symbolError is the error, in half steps, between the current position and the closest symbol.

currentSpeed is the most recent rotational speed for this reel in rpm. This ignores direction so -100 and 100 are both reported as 100 rpm.

On exit the function returns success (TRUE) or failure (FALSE). The failure reason can be determined by calling the function GetLastError().

Example

```
/* get extended status for reel 3 */
XReelStatusEx statusEx;

if ( !board->VendorGetReelStatus( 3, &statusEx ) )
{
       /* error obtaining status */
}
```

5.12 Usage Notes

5.12.1 General

When using the enhanced API it is recommended that customers do not use any of the reel spinning API calls from the standard API. Most of those functions will still work but may leave a reel in between symbols or may overwrite the reel configuration.

In normal operation, reels will always stop exactly at symbol positions. If a reel is stopped between symbols and a request is made to move the reel by a number of symbols then the API will advance the reel in the next symbol position in the given direction and only then move the reel the required number of symbols.

All of the API functions report success or failure when called. It is strongly recommended that the application software checks the success of each call and takes an appropriate action in the event of a failure.

5.12.2 Initialisation

Assuming that the Reel Definition File is present and correct, the typical initialisation sequence would be:

1. Set the file path to the Reel Definition File with a call to *VendorSetReelFilePath()*.

2. For each reel, call *VendorConfigureReel()* passing in the reel type.

3. For each reel, call *VendorSynchroniseReel()*.

At this point the reels are all initialised, synchronised and are at the home position.

5.12.3 Limitations

Reel Definition File size: unlimited
The XML file size is limited only by disk space and PC memory.
The entire XML file is loaded into PC memory before being parsed but, in practice, the size of this file, and therefore the maximum number of supported reels, can be considered unlimited.

Max File Path length: 250 characters.
This is the maximum length of the fully qualified path to the reel definition file, including the filename and extension.
For example, “f:\reels\heber-reel-definitions.xml”

Symbols per reel: 200
This is the number of symbols in one revolution of the reel.

Ramp table pairs per reel: 25.
A ramp table pair comprises one “up” table and one “down” table so this limit means that each reel can have up to 25 different speeds.

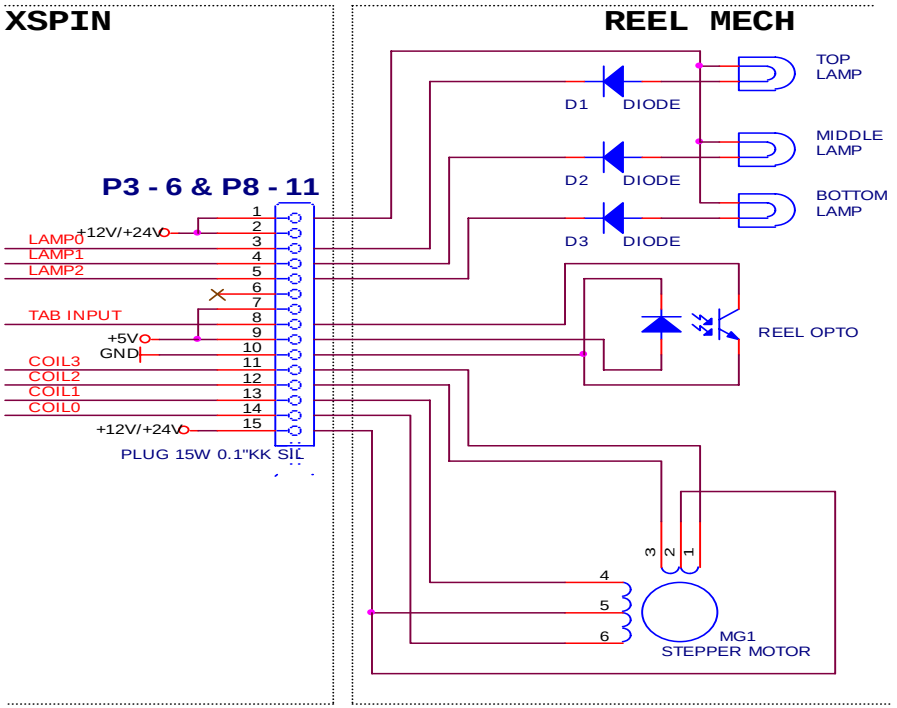
Max Ramp Table Length: 60
This is the maximum number of ramp table entries for each ramp direction in each ramp table pair. So there can be 60 “up” entries and 60 “down” entries per ramp table.

6 ERROR CODES

Code	Definition	Description
0	XREEL_MSG_SUCCESS	Function call performed successfully
1	XREEL_MSG_BOARD_NOT_INITIALISED	A function could not be performed as the board has not yet been initialised
2	XREEL_MSG_INVALID_POINTER	A pointer passed to a function is not valid
3	XREEL_MSG_PARAMETER_OUT_OF_RANGE	A parameter supplied to a function is outside the permitted range for that function
4	XREEL_MSG_INPUT_OUT_OF_RANGE	A parameter relating to a board input is outside the permitted range
5	XREEL_MSG_OUTPUT_OUT_OF_RANGE	A parameter relating to a board output is outside the permitted range
6	XREEL_MSG_INPUT_NOT_AVAILABLE	A parameter relating to a board input is not available to use as it has already been assigned to another function
7	XREEL_MSG_OUTPUT_NOT_AVAILABLE	A parameter relating to a board output is not available to use as it has already been assigned to another function
8	XREEL_MSG_REEL_OUT_OF_RANGE	The reel specified does not exist
9	XREEL_MSG_REEL_NOT_CONFIGURED	An attempt has been made to call a reel function, where that reel has not been fully configured
10	XREEL_MSG_REEL_BUSY	The selected reel is still busy processing a previous request
11	XREEL_MSG_RAMP_TABLE_MISSING	An attempt has been made to call a reel function, that requires the use of a ramp table, prior to the ramp table being supplied
12	XREEL_MSG_RAMP_TABLE_LENGTH_ERROR	The number of entries in a supplied ramp table is not valid.
13	XREEL_MSG_INVALID_STEP_SIZE	The step size parameter passed to a function is outside the permitted range
14	XREEL_MSG_INVALID_DUTY_PERIOD	The total duty cycle duration is outside the permitted range allowed
15	XREEL_MSG_TOO_MANY_STEPS	The parameter relating to the number of steps on a reel is greater than the permitted range
16	XREEL_MSG_NOT_ENOUGH_STEPS	The parameter relating to the number of steps on a reel is smaller than the permitted range
17	XREEL_MSG_USB_TRY_AGAIN	A function call could not be carried out, try and call the function again
18	XREEL_MSG_USB_TIMED_OUT	A timeout event occurred while performing a function call requiring USB communications with a board
19	XREEL_MSG_USB_DEVICE_WRITE_ERROR	A write error occurred while performing a function call requiring USB communications with a board
20	XREEL_MSG_USB_MSG_LENGTH_ERROR	The length of the USB message sent to the board over the USB communications bus is out of the permitted range
21	XREEL_MSG_USB_DEVICE_RESPONSE_ERROR	A device response error has occurred while performing a function call requiring USB communications with a board
22	XREEL_MSG_USB_CMD_NOT_RECOGNISED	The command passed to a board over the USB communications has not been recognised
23	XREEL_MSG_USB_CMD_NOT_SUPPORTED	The command passed to a board over the USB communications is not supported in the current operating mode
24	XREEL_MSG_REEL_IDLE	The stop command of the skill stop feature has been called while the reel is idle.

Code	Definition	Description
25	XREEL_MSG_REEL_INVALID_MODE	The stop command of the skill stop feature has been called while the reel is operating in standard spinning mode.
26	XREEL_MSG_REEL_NOT_SYNCHRONISED	The function requires the reel to be synchronised prior to calling.
27	XREEL_MSG_FILE_NOT_FOUND	The API either can't locate or can't open the Reel Definition File.
28	XREEL_MSG_FILE_PATH_TOO_LONG	The file path of the Reel Definition File is too long.
29	XREEL_MSG_REEL_DATA_NOT_FOUND	The requested reel data cannot be found in the Reel Definition File.
30	XREEL_MSG_ERROR_IN_REEL_DATA	An error has been detected in the reel data.

7 REEL MECHANISM WIRING



Title: XSPIN Reel Wiring			
Size: A	Document Number: 56-20428		Rev: 2
Date: Wednesday, March 04, 2009	Sheet 1 of 1		