FIREFLY

A new Generation of **Video** Control Boards

Firefly X10 - Software User Manual

Document No. 80-17423 Issue 6r2 HEBER LTD

Current Issue: - Issue 6r2 – 1st June 2005

Previous Issue: - Issue 6r1 – 31st May 2005

HEBER LTD

Belvedere Mill
Chalford
Stroud
Gloucestershire
GL6 8NT
England

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

CONTENTS

This page intentionally left blank.

1 INTRODUCTION

This manual, as a software user manual describes the functions available for development purposes.

2 X10 PROCESSING SPEED CONSIDERATIONS

The X10 board is based around an 8051 processor, and therefore has a finite amount of processing power available. Although every effort has been made to shield the end user from the X10 processing constraints, it is important to have an understanding of how different X10 API calls impact on X10 performance, in addition to the fact that certain X10 calls take longer to perform than others.

2.1 The 1ms interrupt

The most important measure of X10 load is the time taken to perform the 1ms interrupt. If 100% of the 1ms interrupt is used then the X10 will certainly become sluggish, and may also become unreliable. An approximate safe proportion of the 1ms interrupt to use is 95%.

Before any X10 API calls have been used and the X10 is idle, the 1ms interrupt load is at 25.2%. This appears to be a high load considering the X10 is “doing nothing”, however the X10 does perform many important operations that are hidden from the user.

We will now look at how different X10 API calls increase the 1ms interrupt. We will only look at X10 functions that significantly consume the interrupt. Functions that consume small amounts of the interrupt (less than 4%) will be ignored.

Please remember that the following values are for guidance only, and provided in the hope that they prove helpful when debugging your software. In “real life” X10 applications, where many X10 functions are working simultaneously, the following interrupt usage data may change somewhat.

2.1.1 Reel Spinning

For each configured X10 reel output, approximately 9.5% of the 1ms interrupt is used. If all 3 reels are configured, 29.6% of the interrupt will be used.

2.1.2 Parallel Hopper Coin Release

When releasing coins from a parallel hopper device by calling the ReleaseParallelHopperCoins() function, approximately 12.8% of the 1ms interrupt is consumed. This drops back to zero once all coins have been successfully released.

2.1.3 Pulsed Inputs

For each configured pulsed input, approximately 6% of the 1ms interrupt will be used.

2.1.4 Cctalk

For each configured cctalk mode 1 device (see chapter 9 for cctalk mode description), 4.8% of the interrupt is used. So if all 4 cctalk devices are configured on port A then 19.2% of the interrupt will be consumed, and if all 4 devices on port B are also used then this value will double to 38.4% of the interrupt.

2.1.5 SPI

During an SPI transaction, the 1ms interrupt increases by 10.4%.

2.2 Slow X10 Functions

The majority of the X10 API calls work at relatively high speeds, limited mainly by USB transfer constraints (4ms per message). However there are certain X10 calls that take longer to complete, and one should be aware of this when designing software to use the X10.

All Security Pipe functions (except those for the random number generator) are relatively slow. This is because the onboard PIC device performs the processing for these operations. Each Security Pipe function has the additional overhead of transmitting the instruction to the PIC, having the PIC perform the required operation and then obtaining the response from the PIC. All PIC transactions occur over a slow I^2C bus.

Reading from and writing to EEPROM is slow due to EEPROM transfer speed constraints. The time taken to perform either a write or a read goes up in a linear fashion depending on the number of bytes to transfer.

When transmitting SPI data using either SendSPI() or SendSEC(), the X10 will actually hang the SPI pipe until the transaction (both transmit and receive) is complete. So the larger the amount of SPI data to be transmitted, the longer the function will take to relinquish the SPI pipe.

3 FIREFLY X10 SYSTEM FUNCTIONS

3.1 Function name: FireFlyUSB()

This function is called automatically when creating a variable of type *FireFlyUSB*. Initialises internal class variables.

3.1.1 Function Prototype

FireFlyUSB()

3.1.2 Programming Considerations

None. No pipes are used for this function.

3.2 Function name: ~FireFlyUSB( )

This is called automatically and cannot be called directly. Cleans up internal class variables in addition to closing the USB device if it is still open.

3.2.1 Function Prototype

~FireFlyUSB( );

3.2.2 Programming Considerations

None. No pipes are used for this function.

3.3 Function name: InitUSBBoard( )

Opens a Firefly X10 device. The boardNumber variable refers 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.

Thus, up to eight Firefly X10 boards can be connected to a single motherboard, although an additional hub would be required to allow this number of physical connections.

Subsequently, all other function calls will be made to this board.

3.3.1 Function Prototype

BOOL InitUSBBoard(BYTE boardNumber);

3.3.2 Programming Considerations

None. No pipes are used for this function.

3.4 Function name: init( )

Opens the only Firefly X10 USB device. This is a simplified version of the previous function. It can be used when there is only one Firefly X10 connected to the system and the address switch setting is to be ignored.

3.4.1 Function Prototype

BOOL init( );

3.4.2 Programming Considerations

Do not use this function if more than one Firefly X10 is connected to the system. No pipes are used for this function.

3.5 Function name: close( )

Closes the USB device.

3.5.1 Function Prototype

BOOL close( );

3.5.2 Programming Considerations

None. No pipes are used for this function.

3.6 Function name: GetProductVersion( )

This function reports the Firefly X10 Development Suite Product version.

3.6.1 Function Prototype

BOOL GetProductVersion(LPBYTE versionProduct);

3.6.2 Programming Considerations

This returns a string of no more than 10 characters including the null terminator.

No pipes are used for this function.

3.7 Function name: GetDllVersion( )

This function reports the version of the Windows DLL that is providing access to the USB device. It may be used to check that the correct DLL has been installed and is being called. It does not require a device handle and can be called without a USB device connected.

3.7.1 Function Prototype

BOOL GetDllVersion(LPBYTE versionDll);

3.7.2 Programming Considerations

This returns a string of no more than 10 characters including the null terminator.

No pipes are used for this function.

3.8 Function name: Get8051Version()

This function reports the version number of the 8051 software. This is downloaded to the Firefly X10 device during enumeration.

3.8.1 Function Prototype

BOOL Get8051Version(LPBYTE version8051);

3.8.2 Programming Considerations

This returns a string of no more than 10 characters including the null terminator.

The Memory pipe is used for this function.

3.9 Function name: GetLastError( )

This function returns the error code for the last function call that failed and updated the error code.

3.9.1 Function Prototype

usbErrorCode GetLastError( );

3.9.2 Return value

A variable of type usbErrorCode that describes the reason for failure.

3.9.3 Programming Considerations

Not all failing function calls have an error code associated with them, in which case the error code returned, if requested, will relate to a previous function call that failed. I/O Pipe Functions

3.10 Function name: GetInputBit()

NOTE: This function is now deprecated and mentioned here only for completeness.

Any of the inputs on the USB board may be read individually using this function call. The bit identification must be one of the following:

- INPUT_BIT_IP0 to INPUT_BIT_IP23
- INPUT_BIT_SW0 to INPUT_BIT_SW7 (These refer to the onboard DIP switch, not the security switch inputs that are read by the PIC).
- INPUT_BIT_CURRENT_SENSE

The current value (1 if high, or 0 if low) will be reported.

3.10.1 Function Prototype

BOOL GetInputBit(usbInputBitId input_bit_id, LPBOOL result);

3.10.2 Programming Considerations

Inputs are continuously sampled every 10ms and must be the same on two successive readings before a change in state is accepted. Therefore, there is a delay of 10 to 20ms between an input level settling to a new state and the change being reported.

The IO pipe is used for this function.

This function is provided for legacy support only. Use the function GetInputs(usbInput * usbInputs) instead, which returns the result into a structure.

3.11 Function name: GetInputs()

NOTE: This function is now deprecated and mentioned here only for completeness.

All of the inputs on the USB board may be read in a single block using this function call. The bytes returned are as follows (in the order in which they are returned):

- INPUT_BIT_IP0-7
- INPUT_BIT_IP8-15
- INPUT_BIT_IP16-23
- INPUT_BIT_SW0-7
- CURRENT_SENSE (0 for no current, 0xff for current detected)

3.11.1 Function Prototype

BOOL GetInputs(LPBYTE inputs);

3.11.2 Programming Considerations

Inputs are continuously sampled every 10ms and must be the same on two successive readings before a change in state is accepted. Therefore, there is a delay of 10 to 20ms between an input level settling to a new state and the change being reported.

The IO pipe is used for this function.

This function is provided for legacy support only. Use the function GetInputs(usbInput * usbInputs) instead, which returns the result into a structure.

3.12 Function name: SetOutputBit()

NOTE: This function is now deprecated and mentioned here only for completeness.

Any of the outputs on the USB board may be set or cleared individually using this function call. The bit identification must be one of the following:

- OUTPUT_BIT_IP0 to OUTPUT_BIT_IP31
- OUTPUT_BIT_AUX0 to OUTPUT_BIT_AUX7

Setting the bit will turn the output ON, clearing the bit will turn the output OFF.

3.12.1 Function Prototype

BOOL SetOutputBit(usbOutputBitId output_bit_id, BOOL bit_state);

3.12.2 Programming Considerations

The outputs are changed as soon as the USB message is received. Messages are sent once every millisecond so there is a delay of up to one millisecond.

Attempts to control outputs that have been assigned to reel stepper motor control will be accepted but ignored.

The IO pipe is used for this function.

3.13 Function name: SetOutputs()

NOTE: This function actually uses ModifyOutputs(), so a direct call to ModifyOutputs() will be slightly more efficient. It will continue to be supported.

All of the outputs on the USB board may be controlled at once using this function call. The bytes set the outputs as follows (in the order in which they are sent):

- OUTPUT_BIT_OP0-7
- OUTPUT_BIT_OP8-15
- OUTPUT_BIT_OP16-23
- OUTPUT_BIT_OP24-32
- OUTPUT_BIT_AUX0-7

Setting bits will turn the related outputs ON, clearing bits will turn the related outputs OFF.

3.13.1 Function Prototype

BOOL SetOutputs(LPBYTE outputs);

3.13.2 Programming Considerations

The outputs are changed as soon as the USB message is received. Messages are sent once every millisecond so there is a delay of up to one millisecond.

Attempts to control outputs that have been assigned to reel stepper motor control will be accepted but ignored.

The IO pipe is used for this function.

This function is provided for legacy support only. Use the function SetOnPeriodOutputs( ) and the related functions (below) to control outputs.

3.14 Function name: GetInputs()

All of the inputs on the USB board may be read in a single block using this function call. The results are returned in a structure of type usbInput, defined in fflyusb.h.

3.14.1 Function Prototype

BOOL GetInputs(usbInput *inputs);

3.14.2 Programming Considerations

Inputs are continuously sampled every 10ms and must be the same on two successive readings before a change in state is accepted. Therefore, there is a delay of 10 to 20ms between an input level settling to a new state and the change being reported.

The IO pipe is used for this function.

3.15 Function name: GetChangedInputs()

Reports the inputs that have changed since the last input read. Note this only works with debounced input reads, and will not work with *GetRawInputs*.

The parameter <changedInputs> is a pointer to a variable of type *usbInput*. Inputs that have changed will be represented by 1's and unchanged inputs will be represented as 0's.

3.15.1 Function Prototype

BOOL GetChangedInputs(usbInput *changedInputs);

3.15.2 Programming Considerations

The IO pipe is used for this function.

3.16 Function name: GetRawInputs()

All of the inputs on the USB board may be read in a single block using this function call. The results are returned in a structure of type usbInput, defined in fflyusb.h. When using this function the inputs are not de-bounced first. The inputs are read instantaneously.

3.16.1 Function Prototype

BOOL GetRawInputs(usbInput *rawInputs);

3.16.2 Programming Considerations

The IO pipe is used for this function.

3.17 Function name: InputMultiplexing()

This function is used to either enable or disable input multiplexing. Once enabled then outputs 12, 13, 14 and 15 are used to strobe the input multiplexer. Each output is sequentially pulsed low for 1mS and the inputs are read and stored at the end of the pulse.

The parameter <input> is used to specify whether input multiplexing should be enabled. It can equal either InputMultiplexDisabled or InputMultiplexEnabled.

3.17.1 Function Prototype

BOOL InputMultiplexing(usbInputMultiplexing input);

3.17.2 Programming Considerations

The IO pipe is used for this function.

3.18 Function name: GetMultiplexedInputs()

This function returns the state of the multiplexed inputs. The data is returned as the following structure (which is defined in 'x10io.h'):

```
 typedef struct
 {
     BYTE byMuxStatus;
     BYTE byMuxInp[ 4 ][ 3 ];
} usbMultiplexedInput;
```

The structure member byMuxStatus indicated the status of the input multiplexer. A non-zero value means that the input multiplexer in enabled, a zero value means that the input multiplexer is disabled.

The structure member byMuxInp contains the status of each input pin for each channel. There are four channels each containing 24 inputs (24 bits = 3 bytes).

Each multiplexed input channel is read every 4mS. There are 4 channels and each corresponds to an output pin 12, 13, 14 or 15. For the first 4ms time slice, output 12 is toggled low while outputs 13, 14, and 15 are set high. All the input pins are read and stored for channel 0. Next output 12 is toggled high and output 13 is toggled low. On the next 1mS pass all inputs pins are read and stored in channel 1. This procedure is repeated for each input channel and is summarised in the table below:

Time	OP12	OP13	OP14	OP15	Input channel read
1mS	**0**	1	1	1	Channel 0
2mS	1	**0**	1	1	Channel 1
3mS	1	1	**0**	1	Channel 2
4mS	1	1	1	**0**	Channel 3

3.18.1 Function Prototype

BOOL GetMultiplexedInputs(usbMultiplexedInput *inputs);

3.18.2 Programming Considerations

The IO pipe is used for this function.

3.19 Function name: ModifyOutputs()

This command allows some outputs to be set, some outputs to be cleared and some outputs left unchanged, all in a single function call. To set an output, set the corresponding bit in parameter “onOutputs”. To clear an output, set the corresponding bit in parameter “offOutputs”. Outputs neither defined to turn on or turn off will remain unchanged.

For example:

```
SetOutputs.byOut[0]  = 0x03;
SetOutputs.byOut[1]  = 0x00;
SetOutputs.byOut[2]  = 0x00;
SetOutputs.byOut[3]  = 0x00;
SetOutputs.byAux   = 0x00;
UnSetOutputs.byOut[0]  = 0x0c;
UnSetOutputs.byOut[1]  = 0x00;
UnSetOutputs.byOut[2]  = 0x00;
UnSetOutputs.byOut[3]  = 0x00;
UnSetOutputs.byAux   = 0x00;

ModifyOutputs( &UnSetOutputs, &SetOutputs);
```

Will turn on the first two outputs and turn off the second two, leaving all the others unchanged.

The command does not apply to dimming: it cannot be used to enable dimming outputs.

3.19.1 Function Prototype

BOOL ModifyOutputs(usbOutput * offOutputs, usbOutput * onOutputs);

3.19.2 Programming Considerations

The IO pipe is used for this function.

3.20 Function name: SetOnPeriodOutputs()

This function determines the state of all outputs during the “on” period of the duty cycle.

Following is a brief description of how output fading works. This is also relevant to the following four function descriptions.

Outputs are driven cyclically every 10ms. Each 10ms cycle is divided again into 10 parts. To turn an output on it is powered for all 10 1ms intervals. To turn an output off it is unpowered for all 10 1ms intervals.

Fading and dimming of outputs is accomplished by strobing. Essentially the more 1ms intervals the output is powered for the brighter it will appear. E.g.:

0.0 = 000000000000000000000000000000
0.1 = 100000000010000000001000000000
0.2 = 110000000011000000001100000000
0.5 = 111110000011111000001111100000
1.0 = 111111111111111111111111111111

At any one time an output can be either off (0.0), on (1.0) or at one other position specified by SetOutputBrightness (0-10) – all outputs that are not on/off will have the same brightness. The one caveat to this is mentioned later.

The SetOutputBrightness function defines the on and off period lengths (which sum to 10). To define an output with this brightness we state whether it is powered during the on/off periods (SetOn/SetOffPeriod ….).

Output State	On period	Off period
Off	0	0
On	1	1
Dimmed	1	0
Inv. Dimmed	0	1

Here “Inv. Dimmed” shows the caveat. When a dimmed output has a brightness of 3, the inv. Dimmed output will have an apparent brightness of 10-3 = 7.

3.20.1 Function Prototype

BOOL SetOnPeriodOutputs(usbOutput onPeriods);

3.20.2 Programming Considerations

The outputs are changed as soon as the USB message is received. Messages are sent once every millisecond so there is a delay of up to one millisecond.

Attempts to control outputs that have been assigned to reel stepper motor control will be accepted but ignored.

The IO pipe is used for this function.

3.21 Function name: SetOffPeriodOutputs()

This function determines the state of all outputs during the “off” period of the duty cycle.

See the SetOnPeriodOutputs() function description above.

3.21.1 Function Prototype

BOOL SetOffPeriodOutputs(usbOutput offPeriods);

3.21.2 Programming Considerations

The outputs are changed as soon as the USB message is received. Messages are sent once every millisecond so there is a delay of up to one millisecond.

Attempts to control outputs that have been assigned to reel stepper motor control will be accepted but ignored.

The IO pipe is used for this function.

3.22 Function name: SetOnPeriodOutputBit()

This function behaves in the same way as SetOnPeriodOutputs() however it allows you to control individual output bits.

The <outputID> parameter specifies the output block to control. The possible values are stored in the usbOutputId structure, which currently contain the possible outputs: USB_OP_0, USB_OP_1, USB_OP_2, USB_OP_3 and USB_OP_AUX.

The <bitNumber> parameter specifies the bit number within the output block to control. These can be bits 0 – 7 for USB_OP_0 – USB_OP_3 and bits 0 – 5 for USB_OP_AUX.

The <bitState> parameter determines the state of the output during the “on” period of the duty cycle. See SetOnPeriodOutputs() for a more complete description.

3.22.1 Function Prototype

BOOL SetOnPeriodOutputBit(usbOutputId outputID, int bitNumber, BOOL bitState);

3.22.2 Programming Considerations

The outputs are changed as soon as the USB message is received. Messages are sent once every millisecond so there is a delay of up to one millisecond.

Attempts to control outputs that have been assigned to reel stepper motor control will be accepted but ignored.

The IO pipe is used for this function.

3.23 Function name: SetOffPeriodOutputBit()

This function determines the state of an individual output bit during the “off” period of the duty cycle.

See the SetOnPeriodOutputBit() function description above.

3.23.1 Function Prototype

BOOL SetOffPeriodOutputBit(usbOutputId outputID, int bitNumber, BOOL bitState);

3.23.2 Programming Considerations

The outputs are changed as soon as the USB message is received. Messages are sent once every millisecond so there is a delay of up to one millisecond.

Attempts to control outputs that have been assigned to reel stepper motor control will be accepted but ignored.

The IO pipe is used for this function.

3.24 Function name: SetOutputBrightness()

This function determines the brightness for output lamps that have previously been defined using the functions SetOnPeriodOutputs, SetOffPeriodOutputs, SetOnPeriodOutputBit and SetOffPeriodOutputBit.

The <brightness> parameter defines the brightness with a range 0 – 10.

3.24.1 Function Prototype

BOOL SetOutputBrightness(BYTE brightness);

3.24.2 Programming Considerations

The IO pipe is used for this function.

3.25 Function name: PulseOutput()

This function is used to pulse a Firefly X10 output and hence determine whether a meter current is detected.

The parameter <outputNumber> is the Firefly X10 output to be pulsed. This value corresponds to Firefly X10 outputs as follows: 0 = OP0, 31 = OP31, 32 = AUX0, 37 = AUX5.

The parameter <pulseDurationMs> specifies the time, in ms, that the output should be pulsed.

This function is non-blocking, hence a call to *PulseOutputResult* should be made to obtain pulse results.

3.25.1 Function Prototype

BOOL PulseOutput(BYTE outputNumber, BYTE pulseDurationMs);

3.25.2 Programming Considerations

The IO pipe is used for this function.

3.26 Function name: PulseOffOutput()

This function is used to pulse a Firefly X10 output and hence determine whether a meter current is detected. This function behaves in the same way as PulseOutput() except with reverse logic levels.

The parameter <outputNumber> is the Firefly X10 output to be pulsed. This value corresponds to Firefly X10 outputs as follows: 0 = OP0, 31 = OP31, 32 = AUX0, 37 = AUX5.

The parameter <pulseDurationMs> specifies the time, in ms, that the output should be pulsed.

This function is non-blocking, hence a call to *PulseOutputResult* should be made to obtain pulse results.

3.26.1 Function Prototype

BOOL PulseOffOutput(BYTE outputNumber, BYTE pulseDurationMs);

3.26.2 Programming Considerations

The IO pipe is used for this function.

3.27 Function name: PulseOutputResult()

This function is used to obtain pulse output results following a call to function *PulseOutput*.

The parameter <timeRemaining> is a pointer to a BYTE that specifies the time, in ms, until the pulse output is complete.

The parameter <pulseComplete> is a pointer to a BOOL that specifies whether or not the pulse output is complete.

The parameter <currentDetected> is a pointer to a BOOL that specifies whether or not meter current was detected.

3.27.1 Function Prototype

```
BOOL PulseOutputResult(        LPBYTE timeRemaining,
                               BOOL *pulseComplete,
                               BOOL *currentDetected );
```

3.27.2 Programming Considerations

The IO pipe is used for this function.

3.28 Function name: ConfigurePulsedInput()

This function is used to configure reading of input pulses on parallel devices, for example coin acceptors. The following functions, described in the following subsections, are used in conjunction with this function: *BeginPulsedInputCheck()*, *EndPulsedInputCheck()*, *ResetPulsedInputCounter()*, *DecrementPulsedInputCounter()* and *ReadPulsedInputCounter()*.

Pulse reading will not begin until a call to *BeginPulsedInputCheck()* is made.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used.

The parameter <pulseLowerTime> specifies the minimum pulse length, in ms, for valid input pulses.

The parameter <inputActiveState> specifies whether the input is active high or low, and can only take the values High or Low.

3.28.1 Function Prototype

```
BOOL ConfigurePulsedInput(      usbInputBitId inputBitID,
                                BYTE pulseLowerTime,
                                ActiveState inputActiveState );
```

3.28.2 Programming Considerations

The IO pipe is used for this function.

3.29 Function name: BeginPulsedInputCheck()

This function begins pulsed input counting on the specified input. A call to *ConfigurePulsedInput()* must be made first.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used.

3.29.1 Function Prototype

```
BOOL BeginPulsedInputCheck( usbInputBitId inputBitID );
```

3.29.2 Programming Considerations

The IO pipe is used for this function.

3.30 Function name: EndPulsedInputCheck()

This function ends pulsed input counting on the specified input. A call to *ConfigurePulsedInput()* must be made first.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used.

3.30.1 Function Prototype

```
BOOL EndPulsedInputCheck( usbInputBitId inputBitID );
```

3.30.2 Programming Considerations

The IO pipe is used for this function.

3.31 Function name: ResetPulsedInputCounter()

This function resets (sets to zero) the pulsed input counter on the specified input. A call to *ConfigurePulsedInput()* must be made first.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used.

3.31.1 Function Prototype

BOOL ResetPulsedInputCounter(usbInputBitId inputBitID);

3.31.2 Programming Considerations

The IO pipe is used for this function.

3.32 Function name: DecrementPulsedInputCounter()

This function decrements the pulsed input counter on the specified input. If the counter is currently zero then no decrement will occur. A call to *ConfigurePulsedInput()* must be made first.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used.

3.32.1 Function Prototype

BOOL DecrementPulsedInputCounter(usbInputBitId inputBitID);

3.32.2 Programming Considerations

The IO pipe is used for this function.

3.33 Function name: ReadPulsedInputCounter()

This function reads the pulsed input counter on the specified input. A call to *ConfigurePulsedInput()* must be made first.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used.

The parameter <inputCounterValue> is a pointer to a BYTE where the returned counter value will be stored.

3.33.1 Function Prototype

BOOL ReadPulsedInputCounter(usbInputBitId inputBitID, BYTE *inputCounterValue);

3.33.2 Programming Considerations

The IO pipe is used for this function.

3.34 Function name: ReleaseParallelHopperCoins()

This function is used to release coins from a parallel hopper device. This is a non-blocking function, so in order to read the release status a call to *GetParallelHopperStatus()* must be made.

This function works by setting a specified output high – this will turn the coin hopper motor on and start to release coins. Simultaneously an input is read counting off pulses – these pulses equate to released coins. If all the required coins are not released, and a specified timeout occurs, it will be assumed that an error has occurred. In this event the output will be set low (turning off the hopper motor) and an error will be returned – see function *GetParallelHopperStatus()*.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used. These pulses equate to released coins.

The parameter <pulseLowerTime> specifies the minimum pulse length, in ms, for valid input pulses.

The parameter <inputActiveState> specifies whether the input is active high or low, and can only take the values High or Low.

The parameter <outputBit> specifies the Firefly X10 output. This will be set high until all required coins are released, or until timeout in which case it is assumed there is a failure.

The parameter <coinTimeout> specifies a timeout, in ms, for coin release. If this timeout occurs with no coin being released, it is assumed a fault has occurred and the output will be set low.

The parameter <coinsToRelease> specifies the number of coins to release.

3.34.1 Function Prototype

```
BOOL ReleaseParallelHopperCoins(      usbInputBitId inputBit,
                                      BYTE pulseLowerTime,
                                      ActiveState inputActiveState
                                      UsbOutputBitId outputBit,
                                      WORD coinTimeout,
                                      BYTE coinsToRelease );
```

3.34.2 Programming Considerations

The IO pipe is used for this function.

3.35 Function name: GetParallelHopperStatus()

This function reads the parallel hopper coin release status after a call to *ReleaseParallelHopperCoins()* has been made.

The parameter <inputBitID> specifies the input bit to use for pulse reading. Inputs 0 to 23 may be used. These pulses equate to released coins.

The parameter <status> is a pointer to a structure of type *ReleaseHopperCoinsStatus*. It can take three values. *Running* implies that coins are still being released and there are no problems. *Failure* implies that the timeout has occurred without a coin being released, and that it has failed. *Success* implies that all required coins have been successfully released.

The parameter <coinsReleased> is a pointer to a BYTE which will contain the number of coins that have currently been released.

3.35.1 Function Prototype

```
BOOL GetParallelHopperStatus(  usbInputBitId inputBit,
                               ReleaseHopperCoinsStatus *status,
                               BYTE *coinsReleased );
```

3.35.2 Programming Considerations

The IO pipe is used for this function.

3.36 Function name: ConfigureReels()

Note: This function is now obsolete. The Function **ConfigureReelsEx()** (documented in section 3.37) should be used instead.

This function must be called if reels are to be controlled by the X10 board. The parameters determine the number of reels fitted (up to 3, identified as reel 0, reel 1 and reel 2), the number of positions per turn and the number of steps per symbol. (Note that one step is two positions when full stepping, but only one position when half stepping).

At power up, the USB board will initialise with no reels fitted until this function has been called. The function may be called with the number of reels set to 0 to disable the reel function and return associated outputs to their normal operation.

If, for example, only one reel is configured, then the outputs that would be associated with the two unused reel positions will behave as normal outputs. The system is, therefore, quite flexible.

Note, however, there is no flexibility in allocation of particular outputs to particular reels: if one reel is configured, it will use outputs OP16/17/18/19 and input IP6. The second reel, if configured, will use outputs OP20/21/22/23 and input IP7. The third reel, if configured, will use outputs OP24/25/26/27 and input IP8.

Function calls such as SetOutputs(), will not corrupt outputs configured for use by reels if they try to access them.

Function calls such as GetInputs(), will read and return inputs correctly, whether configured for use with reels or not.

The <positionsPerTurn> variable is the number of half-steps required to turn the reel through one turn. It is used by the synchronisation error checking software.

The <stepsPerSymbol> variable is not currently used and is reserved for future enhancements.

3.36.1 Function Prototype

```
BOOL ConfigureReels(    BYTE numberOfReels,
                        BYTE halfStepsPerTurn,
                        BYTE stepsPerSymbol );
```

3.36.2 Programming Considerations

This function must be called before reel outputs can be used.

This function is now obsolete because the number of half-steps per turn is stored as a BYTE.
Please use ConfigureReelsEx() instead, because the half-steps per turn are stored as a WORD – this accommodates 200 step reels.

The IO pipe is used for this function.

3.37 Function name: ConfigureReelsEx()

This function behaves in exactly the same way as ConfigureReels(), however the number of half steps per turn is now stored in a WORD instead of a BYTE. This allows for 200 step reels.

It is recommended to use this function instead of ConfigureReels().

3.37.1 Function Prototype

```
BOOL ConfigureReelsEx(BYTE numberOfReels,
                      WORD halfStepsPerTurn,
                      BYTE stepsPerSymbol );
```

3.37.2 Programming Considerations

This function must be called before reel outputs can be used.

The IO pipe is used for this function.

3.38 Function name: SpinReels()

This function will spin a reel <reelNumber> through a number of steps <steps>. Each step size and direction is determined by <directionAndStepSize>. The allowed values for <directionAndStepSize> are –2 or 2 if full-stepping is required, or –1 or 1 if half-stepping is required.

This command cannot be used until the ramp up and ramp down tables have been defined for the reel. Note that each reel has its own ramp up and ramp down tables. The number of steps must be greater than the number of steps that are required to perform a complete ramp up and ramp down, otherwise an error will report that not enough steps were requested. The minimum number of steps that can be moved is:

"(number of steps in the ramp up table) + (number of steps in the ramp down table) – 1"

This allows a single step (or single half-step) nudge by using ramp up and ramp down tables with one entry and a spin command of one step. See the sections below describing the ramp table commands for more information on ramps.

The maximum number of steps that can be moved is 32767.

3.38.1 Function Prototype

```
BOOL SpinReels( BYTE reelNumber, BYTE directionAndStepSize, WORD steps );
```

3.38.2 Programming Considerations

Do not call this function without defining the ramp up and ramp down tables for the reel first.

Ensure that the number of steps requested will allow the ramp tables to be carried out, otherwise an error will be reported and the reel will not move.

The IO pipe is used for this function.

3.39 Function name: SpinRampUp()

This command must be issued for reel number <reelNumber> before spinning a reel for the first time. It defines how the reel will be ramped up. Each reel needs a ramp table (which may be different for each reel). Without a ramp table, reel behaviour will be unpredictable.

The variable <rampUpTable> must be of structure type <usbRampTable>. 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). Note that 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.

3.39.1 Function Prototype

BOOL SpinRampUp(BYTE reelNumber, usbRampTable rampUpTable);

3.39.2 Programming Considerations

Do not exceed the maximum table length. Remember that the last entry in the table also defines the spin speed after the ramp up.

The IO pipe is used for this function.

3.40 Function name: SpinRampDown()

This command must be issued for reel number <reelNumber> before spinning a reel for the first time. It defines how the reel will be ramped down. Each reel needs a ramp table (which may be different for each reel). Without a ramp table, reel behaviour will be unpredictable.

The variable <rampDownTable> must be of structure type <usbRampTable>. 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). Note that 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.

3.40.1 Function Prototype

BOOL SpinRampDown(BYTE reelNumber, usbRampTable rampDownTable);

3.40.2 Programming Considerations

The IO pipe is used for this function.

3.41 Function name: SetDutyCycle()

This function programs the duty cycle timing for a specified reel. The duty cycle occurs when the reel is at rest.

The argument <reelNumber> specifies the reel number to configure (0-2).

The argument <offPeriod> specifies the time (in ms) for the off period of the duty cycle.

The argument <onPeriod> specifies the time (in ms) for the on period of the duty cycle.

If the user does not explicitly execute this function, then a default duty cycle of 5ms off/5ms on will be used.

3.41.1 Function Prototype

BOOL SetDutyCycle(BYTE reelNumber, BYTE offPeriod, BYTE onPeriod);

3.41.2 Programming Considerations

Please ensure that the combined vlaue of offPeriod and onPeriod does not exceed 255.

The IO pipe is used for this function.

3.42 Function name: GetReelStatus()

Note: This function is now obsolete. The Function **GetReelStatusEx()** (documented in section 3.43) should be used instead.

This function returns the reel status for all reels, even only some of the reel outputs are configured as reels. The data returned in the <status> structure describes, for each reel, the following:

- The current reel position (0-255)
- The last reel position error value (-128 to +127), which is the difference between actual position and expected position when the opto-detector was last passed
- The reel busy state, which will be TRUE if busy and the last spin request has not yet been completed
- The 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().

3.42.1 Function Prototype

BOOL GetReelStatus(usbReelStatus * status);

3.42.2 Programming Considerations

Use this function regularly to check that reels remain synchronised.

This function is now obsolete because the reel position and error are stored as BYTE’s. Please use GetReelStatusEx() instead, because the reel position and error are stored as WORD’s – this accommodates 200 step reels.

The IO pipe is used for this function.

3.43 Function name: GetReelStatusEx()

This function returns the reel status for all reels, even if not all reels are configured active. The data returned in the <status> structure describes, for each reel, the following:

- 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 opto-detector was last passed
- The reel busy state, which will be TRUE if busy and the last spin request has not yet been completed
- The 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().

This function is now recommended instead of GetReelStatus() because the reel position and error are now stored as WORD’s instead of BYTE’s. This accommodates 200 step reels.

3.43.1 Function Prototype

BOOL GetReelStatusEx(ReelStatusEx * status);

3.43.2 Programming Considerations

Use this function regularly to check that reels remain synchronised.

The IO pipe is used for this function.

3.44 Function name: ReelSynchroniseEnable()

This function is used to synchronise the reel position counter. A synchronisation flag (state) is used to control initialisation of the reel position counter.

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().

3.44.1 Function Prototype

BOOL ReelSynchroniseEnable(BYTE reelNumber);

3.44.2 Programming Considerations

The reels must be spun through at least one full turn after calling this function, to ensure that the opto-detector is passed.

Note that reels will only synchronise when spinning **forwards** (positive direction/step size value) past the opto-detector.

The IO pipe is used for this function.

4 SERIAL PIPE A AND SERIAL PIPE B FUNCTIONS

4.1 Function name: SetConfig()

This function configures the requested serial port (PORT_A or PORT_B).

The <port> parameter determines whether the configuration data is for Port A or Port B.

The <config> parameter implements portions of the windows serial DCB structure. The implemented structure variables are: *fOutxCtsFlow*, *fRtsControl*, *fParity* and *Parity*.

fOutxCtsFlow indicates whether the CTS (clear-to-send) signal is monitored for output flow control. If this member is TRUE and CTS is turned off, output is suspended until CTS is sent again.

fParity indicates whether parity checking is enabled. If this member is TRUE, parity checking is performed and errors are reported using the function *GetParityErrors()*. The parity method is set using the member *Parity*, and this can be set to the following values: NOPARITY (no parity), ODDPARITY (odd parity) and EVENPARITY (even parity).

fRtsControl controls how handshaking is handled. This parameter is only relevant when in RS232 mode. The possible values for this member are described below:

Value	Description
RTS_CONTROL_DISABLE	Disables the RTS line when the device is opened and leaves it disabled.
RTS_CONTROL_ENABLE	Enables the RTS line when the device is opened and leaves it on.
RTS_CONTROL_HANDSHAKE	Enables RTS handshaking. The driver raises the RTS line when the “type-ahead” (input) buffer is less than one-half full and lowers the RTS line when the buffer is more than three-quarters full. If handshaking is enabled, it is an error for the application to adjust the line by using the EscapeCommFunction function. In this implementation, it is treated as RTS_CONTROL_TOGGLE below.
RTS_CONTROL_TOGGLE	Specifies that the RTS line will be high if bytes are available for transmission. After all buffered bytes have been sent, the RTS line will be low.

The <type> parameter must be either PORT_RS232, PORT_RS232_POLLED, PORT_CCTALK or PORT_CCTALK_MODE1.

Port B uses TTL levels and signal polarities and has no handshake lines. Therefore, Port B should be set to RTS_CONTROL_DISABLE when port-type is PORT_RS232 or PORT_RS232_POLLED.

The idle condition for port-type PORT_RS232 and PORT_RS232_POLLED is a “1” or marking condition. For Port A, this will be –10V. For Port B, this will be +5V (i.e. TTL).

Note that for PCB issues before issue 6, Port B was identical to Port A and used RS232 levels, not TTL levels.

4.1.1 Function Prototype

BOOL SetConfig(usbSerialPort port, LPDCB config, usbPortType type);

4.1.2 Programming Considerations

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

4.2 Function name: Send()

This function sends data for serial transmission to the requested port (PORT_A or PORT_B). The function is “blocking” and will not return control until the transmission has completed or has failed. Therefore the delay at low baud rates could be lengthy.

The length parameter indicates how many bytes must be transmitted.

A buffered (“non-blocking”) version of this function that allows transmission of data as a background task may be developed in future.

4.2.1 Function Prototype

```
BOOL Send( usbSerialPort port, LPBYTE data, UINT length );
```

4.2.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is not available in RS232 Polled Mode or cctalk Mode 1.

4.3 Function name: Receive()

This function requests any data that has been received on the requested serial port (PORT_A or PORT_B).

The length parameter indicates how many bytes were received.

4.3.1 Function Prototype

```
BOOL Receive( usbSerialPort port, LPBYTE data, LPUINT length );
```

4.3.2 Programming Considerations

Each function call will return a maximum of 62 bytes across the USB interface. However, the API will read packets until the receive buffer in Firefly X10 is empty. The Firefly X10 receive buffer is 600 bytes. Several calls may be required to return all of the data received. If the buffer overflows, approximately 600 bytes will be lost. This is because the buffer is circular.

The function SetConfig() must be called first to ensure that the serial port is correctly configured.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is not available in RS232 Polled Mode or cctalk Mode 1.

4.4 Function name: ReceiveByteWithTimestamp()

If a byte has been received on the requested serial port, the byte is returned (rxByte) along with the corresponding time, in milliseconds, since the last received byte (interval). The interval time is clamped at 255 milliseconds.

The ‘received’ parameter is TRUE if a byte has been returned otherwise FALSE.

4.4.1 Function Prototype

```
BOOL ReceiveByteWithTimestamp(       usbSerialPort port,
                                     LPBYTE rxByte,
                                     LPBYTE interval,
                                     BOOL *received );
```

4.4.2 Programming Considerations

Only one byte is returned per call, therefore several calls may be required to return all of the data queued on the serial port.

The function SetConfig() must be called first to ensure that the serial port is correctly configured.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is not available in RS232 Polled Mode or cctalk Mode 1.

4.5 Function name: SetTimeoutMessage()

This function allows a time out to be defined and an associated message that should be sent if the time-out expires.

The ‘message’ parameter defines the message to send upon timeout, and the ‘numBytes’ parameter defines the number of bytes in the message (1 – 60).

The ‘timeout’ parameter is the timeout in seconds. This has a range of 0 to 7.65 seconds with a resolution of 30ms.

4.5.1 Function Prototype

```
BOOL SetTimeoutMessage(     usbSerialPort port,
                            LPBYTE message,
                            BYTE numBytes,
                            float timeout );
```

4.5.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is not available in RS232 Polled Mode or cctalk Mode 1.

4.6 Function name: GetParityErrors()

This function returns the number of bytes received with a parity error. This value is reset after it is returned.

The ‘parityErrors’ parameter is a pointer to a WORD where the parity error count will be returned.

4.6.1 Function Prototype

```
BOOL GetParityErrors( usbSerialPort port, LPWORD parityErrors );
```

4.6.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

4.7 Function name: ConfigureCCTalkPort()

This function configures a device on a port (A or B) for cctalk Mode 1 operation. For a more detailed description of cctalk Mode 1, please refer to cctalk explanation describe further in this manual..

It is possible to set up to 8 cctalk devices on a Firefly X10: 4 on Port A and 4 on Port B. Once a Port has been set up for cctalk Mode 1 operation, only the following serial API functions are available for that port:

- ConfigureCCTalkPort()
- EmptyPolledBuffer()
- ReceivePolledMessage()
- DeletePolledMessage()

It is possible to call SetConfig() to re-configure the port to RS232/RS232 Polled/cctalk Mode 0 operation if required.

Two parameters are required for this function:

The <port> parameter determines whether the configuration data is for Port A or Port B.

The <cctalkConfig> parameter is a pointer to a structure of type CCTalkConfig containing cctalk Mode 1 configuration data. This is what the structure looks like:

```
typedef struct
{
        BYTE device_number;
        PollMethod method;
        BYTE next_trigger_device;
        BYTE poll_retry_count;
        int polling_interval;
        WORD max_response_time;
        BYTE min_buffer_space;
        BYTE poll_msg[MAX_POLL_MSG_LENGTH];
        BYTE inhibit_msg[MAX_POLL_MSG_LENGTH];
} CCTalkConfig;
```

The <device_number> variable specifies the device number for the device on the current port. This can take the values 0 – 3 thus enabling 4 devices.

The <method> variable specifies how the device is polled. It can take four values: *Repeated, Triggered*, *Once* and *Disabled*. The poll method must be set to *Repeated* for automatic polling. The *Disabled* option shows that this device is not present. The *Once* option is a special case, used to send a single message to a specific device. The *Triggered* method is used if this device is triggered immediately after a device configured to *Once* or *Repeated* method. A *Triggered* device will not send out messages by itself, only if it is set to poll immediately after another device – see next paragraph.

The <next_trigger_device> variable specifies the next device to trigger immediately after this device has polled. It can take the value 0 – 3. The device that will be triggered after this device must have its <method> set to *Triggered*. If no device is to be triggered after this device is polled then set this variable to *NO_TRIGGER*.

The <poll_retry_count> parameter specifies the maximum number of retries, once a response timeout has occurred, before sending an inhibit message. This can range from 0 to 255.

The <polling_interval> variable is the time, in ms, between polls when the <method> variable is set to *Repeated*. It is also the time until the single message is sent when <method> is set to *Once*. This variable can be in the range of 10 – 2550ms, but it will be rounded down to the nearest 10ms multiple.

The <max_response_time> variable specifies the maximum time, in ms, to wait for a response from the device. It can be in the range of 0 – 65535ms. If no response is obtained within this time period then an *Inhibit* message is sent to the device and the internal Firefly X10 variable <method> is set to *Disabled*.

The <min_buffer_space> variable specifies the minimum amount of receive buffer space allowed, below which an inhibit message will automatically be sent and polling stopped. The buffer is stored in battery-backed “xdata” memory space on the Firefly X10 and will be available after power loss. At this time the buffer space available for each Cctalk Device is 600 bytes.

The <poll_msg> variable is the polling message and can be up to 25 bytes long. Please note that the message length is defined in the actual message. The message byte is in position 1 for cctalk. This value does not include the 5 header and checksum bytes.

The <inhibit_msg> variable is the inhibit message and can be up to 25 bytes long. Please note that the message length is defined in the actual message. The message byte is in position 1 for cctalk. This value does not include the 5 header and checksum bytes.

Once these parameters have been defined, automatic polling will commence if <method> is set to *Repeated*. All received data (excluding local echo) will be stored in a buffer for the device. The buffer is 600 bytes long. The buffer may be read using ReceivePolledMessage() and will include a status byte indicating whether the device has been inhibited.

Polling stops automatically once the space left in the buffer falls below a certain level. It also stops if a response timeout occurs and the maximum number of retries has been reached, as defined in <poll_retry_count>. In either of these situations an inhibit message will be sent. If the device has been inhibited polling can be restarted as follows:

1. Sending a single message re-enabling the cctalk device.
2. Re-starting polling with another call to ConfigureCCTalkPort().

Note that (1) above is needed because an inhibit message will have been sent to the device.

If a device is configured for polling and that device triggers another device, and from that possibly more devices, there are some points to be aware of. If one of the devices becomes inhibited, or if it is reconfigured with the *Disabled* option, then that device and all devices triggered after it will cease polling. If it is reconfigured to the *Once* option, it will poll once and all other devices triggered after it once.

4.7.1 Function Prototype

BOOL ConfigureCCTalkPort(usbSerialPort port, CCTalkConfig *cctalkConfig);

4.7.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured and set to PORT_CCTALK_MODE1 operation.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is only available in cctalk Mode 1.

4.8 Function name: ConfigureRS232Poll()

This function configures a device on a port (A or B) for RS232 Polled operation.

RS232 Polled Mode is very similar to cctalk Mode 1 but with several differences:

- The RS232 hardware is used instead of cctalk.
- It is possible to define whether or not local echo suppression is implemented.
- This is a more generic driver in that the message length byte is not set and can be defined, along with an offset byte to obtain the proper message length.

Apart from these differences, operation is identical to cctalk Mode 1. Please read the section ConfigureCCTalkPort() for more details.

It is possible to set up to 8 polled RS232 devices on a Firefly X10: 4 on Port A and 4 on Port B. Once a Port has been set up for RS232 Polled operation, only the following serial API functions are available for that port:

- ConfigureRS232Poll()
- EmptyPolledBuffer()
- ReceivePolledMessage()
- DeletePolledMessage()

It is possible to call SetConfig() to re-configure the port to RS232/cctalk Mode 0/cctalk Mode 1 operation if required.

Two parameters are required for this function:

The <port> parameter determines whether the configuration data is for Port A or Port B.

The <pollConfig> parameter is a pointer to a structure of type RS232PollConfig containing RS232 Polled configuration data. This is what the structure looks like:

```
typedef struct
{
       BYTE device_number;
       PollMethod method;
       BYTE next_trigger_device;
       BYTE poll_retry_count;
       int polling_interval;
       BOOL remove_local_echo;
       BYTE length_byte_offset;
       BYTE add_to_length_byte;
       WORD max_response_time;
       BYTE min_buffer_space;
       BYTE poll_msg[MAX_POLL_MSG_LENGTH];
       BYTE inhibit_msg[MAX_POLL_MSG_LENGTH];
} RS232PollConfig;
```

The variables <device_number>, <method>, <next_trigger_device>, <poll_retry_count>, <polling_interval>, <max_response_time>, <min_buffer_space>, <poll_msg> and <inhibit_msg> are identical to those used in cctalk Mode 1 operation. Please read the section ConfigureCCTalkPort() for more details.

The <remove_local_echo> variable specifies whether or not local echo is removed. This option should only be used in situations where local echo will arise. Otherwise, loss of valid received data may occur.

The <length_byte_offset> variable specifies where in the message packet the length byte is located. For example if located in the first byte then this would be 0. Please note that only single length bytes are allowed.

The <add_to_length_byte> variable specifies a value to add onto the value stored in the length byte to obtain the proper message length. This is used in protocols where the message contains header and checksum bytes that are ignored in the message length byte.

4.8.1 Function Prototype

BOOL ConfigureRS232Poll(usbSerialPort port, RS232PollConfig *pollConfig);

4.8.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured and set to PORT_CCTALK_MODE1 operation.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is only available in RS232 Polled Mode.

4.9 Function name: EmptyPolledBuffer()

This function empties the receive buffer for the specified device.

Two parameters are required for this function:

The <port> parameter determines whether Port A or Port B is used.

The <deviceNumber> parameter specifies the device number and can take a value 0 – 3.

4.9.1 Function Prototype

```
BOOL EmptyPolledBuffer( usbSerialPort port, BYTE deviceNumber );
```

4.9.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured and set to either PORT_RS232_POLLED or PORT_CCTALK_MODE1 operation. Also a call to ConfigureCCTalkPort() or ConfigureRS232Poll() must be made to configure this device.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is only available in RS232 Polled Mode or cctalk Mode 1.

4.10 Function name: ReceivePolledMessage()

This function returns a received message stored in the buffer. The buffer is a FIFO (first in, first out) and is 600 bytes long. The message remains in the buffer until a call to DeletePolledMessage() is made. This is required to preserve the message in case power loss occurs.

The parameters for this function are described as follows:

The <port> parameter determines whether Port A or Port B is used.

The <deviceNumber> parameter specifies the device number and can take a value 0 – 3.

The <data> parameter is a pointer to a buffer where the received message will be stored.

The <length> parameter is a pointer to an *unsigned int* that will return the number of bytes in the received message.

The <inhibited> parameter is a pointer to a *BOOL* that specifies whether the device has been inhibited.

4.10.1 Function Prototype

```
BOOL ReceivePolledMessage(   usbSerialPort port,
                             BYTE deviceNumber,
                             LPBYTE data,
                             LPUINT length,
                             BOOL *inhibited );
```

4.10.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured and set to either PORT_RS232_POLLED or PORT_CCTALK_MODE1 operation. Also a call to ConfigureCCTalkPort() or ConfigureRS232Poll() must be made to configure this device.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is only available in RS232 Polled Mode or cctalk Mode 1.

4.11 Function name: DeletePolledMessage()

This function deletes the first message stored in the buffer.

Two parameters are required for this function:

The <port> parameter determines whether Port A or Port B is used.

The <deviceNumber> parameter specifies the device number and can take a value 0 – 3.

4.11.1 Function Prototype

BOOL DeletePolledMessage(usbSerialPort port, BYTE deviceNumber);

4.11.2 Programming Considerations

The function SetConfig() must be called first to ensure that the serial port is correctly configured and set to either PORT_RS232_POLLED or PORT_CCTALK_MODE1 operation. Also a call to ConfigureCCTalkPort() or ConfigureRS232Poll() must be made to configure this device.

Depending on the Serial Port, either the SERIAL_A or SERIAL_B pipe is used.

This function is only available in RS232 Polled Mode or cctalk Mode 1.

5 SPI PIPE FUNCTIONS

5.1 Function name: EnableSPI()

This function initialises the SPI (Serial Peripheral Interface) protocol. A call must be made to this function before using any SPI commands.

The SPI protocol uses two Firefly X10 outputs for device communications: OP0 for the clock and OP1 for data output. Once the SPI protocol is enabled then these outputs can only be used for SPI commands. A call to DisableSPI() must be made to free up these outputs. The Firefly X10 input IP18 is also used as a data input from the SPI device.

5.1.1 Function Prototype

```
BOOL EnableSPI( void );
```

5.1.2 Programming Considerations

The SPI pipe is used for this function.

5.2 Function name: DisableSPI()

This function disables the SPI protocol. This releases the Firefly X10 SPI outputs OP0 and OP1 for use by other functions.

5.2.1 Function Prototype

```
BOOL DisableSPI( void );
```

5.2.2 Programming Considerations

The SPI pipe is used for this function.

5.3 Function name: SendSPI()

This function provides communications to a SPI device. Please note that presently only SPI mode 2 is supported.

The <numberOfTxBits> parameter defines the number of message bits to send.

The <txMessage> parameter is an array of bytes containing the message to send.

The <waitTimeMs> parameter defines the time, in ms, after the message has been send that a response from the device should be expected.

The <numberOfRxBits> parameter defines the number of bits expected back from the SPI device.

The <rxMessage> parameter is an array of bytes where the received message will be stored.

5.3.1 Function Prototype

```
BOOL SendSPI(  BYTE numberOfTxBits,
               LPBYTE txMessage,
               BYTE waitTimeMs,
               BYTE numberOfRxBits,
               LPBYTE rxMessage );
```

5.3.2 Programming Considerations

The SPI protocol must be enabled, using EnableSPI(), before calling this function.

The SPI pipe is used for this function.

5.4 Function name: SendSEC()

This function provides communications to a SEC (Starpoint Electronic Counter) device. This function handles all message formatting and checksum creation automatically.

The <command> parameter is a code that defines the type of message to send.

The <id> parameter is the index number of the message being sent. This number is generated by the host machine and starts at 00h. It is incremented every time a new message is sent, up to a value of FFh after which it rolls over to 00h. This ID is used to determine which lost messages may require re-sending by the host.

The <numberOfTxBytes> parameter defines the number of message bytes to send to the SEC device. This number should not include the command, id or checksum. It should only define the number of data bytes to send.

The <txMessage> parameter is an array of bytes containing the message to send.

The <waitTimeMs> parameter defines the time, in ms, after the message has been send that a response from the device should be expected.

The <numberOfRxBytes> parameter defines the number of bytes expected back from the SEC device. This number must only equal the number of data bytes expected back from the device, not the command, id and checksum confirmation bytes.

The <rxMessage> parameter is an array of bytes where the received message will be stored. Note that this will contain the complete received message including checksum etc.

5.4.1 Function Prototype

```
BOOL SendSEC( BYTE command,
              BYTE id,
              BYTE numberOfTxBytes,
              LPBYTE txMessage,
              BYTE waitTimeMs,
              BYTE numberOfRxBytes,
              LPBYTE rxMessage );
```

5.4.2 Programming Considerations

The SPI protocol must be enabled, using EnableSPI(), before calling this function.

The SPI pipe is used for this function.

6 MEMORY PIPE FUNCTIONS

6.1 Function name: CheckEEPROM()

This function attempts to detect the currently fitted EEPROM. It can differentiate between the following devices: 24LC01, 24LC02, 24LC04, 24LC08, 24LC16, 24LC32, 24LC64, 24LC128, 24LC256 and 24LC512.

The function requires a single parameter, <eepromConfig>, which is a pointer to a structure of type X10EEPROM. This function will populate the structure as reproduced below:

```
typedef struct
{
        BYTE pageSize;            ( e.g. for 24LC04 value = 16 )
        BYTE numAddressBytes;     ( e.g. for 24LC04 value = 1 )
        WORD maxAddress;          ( e.g. for 24LC04 value = 511 )
        BYTE i2cAddress;          ( e.g. for 24LC04 value = 0x50 )
} X10EEPROM;
```

The <pageSize> variable specifies the EEPROM page size in bytes.

The <numAddressBytes> variable specifies the number of address bytes on the EEPROM. This can either be 1 or 2.

The <maxAddress> variable specifies the maximum available EEPROM address.

The <i2cAddress> variable specifies the EEPROM i2c address.

Calls to CheckEEPROM() and ConfigureEEPROM() are automatically made during X10 initialisation in an attempt to detect and configure the currently fitted EEPROM.

6.1.1 Function Prototype

BOOL CheckEEPROM(X10EEPROM *eepromConfig);

6.1.2 Programming Considerations

The MEMORY pipe is used for this function.

6.2 Function name: ConfigureEEPROM()

This function allows the user to override automatic EEPROM detection (see CheckEEPROM( )).

The function requires a single parameter, <eepromConfig>, which is a pointer to a structure of type X10EEPROM. Please see the description for CheckEEPROM() for the definition of the structure X10EEPROM.

Calls to CheckEEPROM() and ConfigureEEPROM() are automatically made during X10 initialisation in an attempt to detect and configure the currently fitted EEPROM.

6.2.1 Function Prototype

BOOL ConfigureEEPROM(X10EEPROM *eepromConfig);

6.2.2 Programming Considerations

The MEMORY pipe is used for this function.

6.3 Function name: ReadEEPROM()

This function reads a block of data from EEPROM.

The <address> parameter specifies the start address to read from.

The <data> parameter is a pointer to a block of data in which the EEPROM data will be written. The user must allocate this memory before calling this function. The amount of memory allocated must also be at least as much as specified in the <totalLength> parameter.

The <totalLength> parameter specifies the number of bytes to read from EEPROM.

6.3.1 Function Prototype

BOOL ReadEEPROM(WORD address, LPBYTE data, UINT totalLength);

6.3.2 Programming Considerations

The first 7 bytes of the EEPROM are reserved for the USB device and should not be read. Otherwise the useable address space is entirely dependent on the EEPROM chip fitted. Currently the following EEPROM types are supported: LC00, LC04 and LC08.

See the section on Enumeration for more information regarding the first 7 locations.

The MEMORY pipe is used for this function.

6.4 Function name: WriteEEPROM()

This function writes a block of memory to the EEPROM.

The <address> parameter specifies the start address to write to.

The <data> parameter is a pointer to a block of source data that will be written to EEPROM.

The <totalLength> parameter specifies the number of bytes to write to the EEPROM.

6.4.1 Function Prototype

BOOL WriteEEPROM(WORD address, LPBYTE data, UINT totalLength);

6.4.2 Programming Considerations

The first 7 bytes of the EEPROM are reserved for the USB device and should not be written to. If they are the device will not function correctly. Otherwise the useable address space is entirely dependent on the EEPROM chip fitted. Currently the following EEPROM types are supported: LC00, LC04 and LC08.

See the section on Enumeration for more information regarding the first 7 locations.

The MEMORY pipe is used for this function.

7 SRAM PIPE FUNCTIONS

7.1 Function name: ReadSRAM()

This function reads a block of data from SRAM.

The <address> parameter specifies the start address to read from.

The <data> parameter is a pointer to a block of data in which the SRAM data will be written. The user must allocate this memory before calling this function. The amount of memory allocated must also be at least as much as specified in the <totalLength> parameter.

The <totalLength> parameter specifies the number of bytes to read from SRAM.

7.1.1 Function Prototype

```
BOOL ReadSRAM( WORD address, LPBYTE data, UINT totalLength );
```

7.1.2 Programming Considerations

It is only possible to read from addresses 0000h to 7E00h. If an attempt is made to read outside of this range then a read error will be returned.

The SRAM pipe is used for this function.

7.2 Function name: WriteSRAM()

This function writes a block of memory to the SRAM.

The <address> parameter specifies the start address to write to.

The <data> parameter is a pointer to a block of source data that will be written to SRAM.

The <totalLength> parameter specifies the number of bytes to write to the SRAM.

7.2.1 Function Prototype

```
BOOL WriteSRAM( WORD address, LPBYTE data, UINT totalLength );
```

7.2.2 Programming Considerations

It is only possible to write to addresses 0000h to 7E00h. If an attempt is made to write outside of this range then a write error will be returned.

The SRAM pipe is used for this function.

8 SECURITY PIPE FUNCTIONS

8.1 Function name: GetPICVersion()

This function reports the version of Firefly X10 PIC (security device) firmware.

8.1.1 Function Prototype

BOOL GetPICVersion(LPBYTE versionPIC);

8.1.2 Programming Considerations

This returns a string of no more than 10 characters including the null terminator.

The Security pipe is used for this function.

8.2 Function name: GetPICSerialNumber()

This function reports the serial number of the PIC. Each PIC supplied by Heber Ltd on the Firefly X10 board is blank. If the Serial number is not set by the user at anytime, using this function will return a random 8 character string.
The security of the PIC is controlled by unlockio.lib and this serial number is not related to the security.

8.2.1 Function Prototype

BOOL GetPICSerialNumber(LPBYTE serialNumberPIC);

8.2.2 Programming Considerations

This returns a string of no more than 9 characters including the null terminator.

The Security pipe is used for this function.

8.3 Function name: SetPICSerialNumber()

This function sets the serial number for the PIC. The serial number is 8 characters long. This can only be used once. The security of the PIC is controlled by unlockio.lib and this serial number is not related to the security.

8.3.1 Function Prototype

BOOL SetPICSerialNumber(LPBYTE serialNumberPIC);

8.3.2 Programming Considerations

The Security pipe is used for this function.

8.4 Function name: SetClock()

This function sets an internal clock to any required value. The clock will then be updated every second. The <time> parameter is a 32-bit value in which to specify the time.

8.4.1 Function Prototype

BOOL SetClock(DWORD time);

8.4.2 Programming Considerations

The SECURITY pipe is used for this function.

8.5 Function name: GetClock()

This function returns the time, in seconds, as set using SetClock(), plus the number of seconds that have elapsed since.

The <time> parameter is a pointer to a DWORD where the 32-bit time will be placed.

8.5.1 Function Prototype

BOOL GetClock(LPDWORD time);

8.5.2 Programming Considerations

The SECURITY pipe is used for this function.

8.6 Function name: NextSecuritySwitchRead()

Four security switch inputs are continuously monitored. Each time they change state, the new state is stored in a circular buffer with a time stamp from the real time clock. The time stamp format is the same as the real time clock format. The switch locations are:

- Switch 0 = Bit 0
- Switch 1 = Bit 1
- Switch 2 = Bit 2
- Switch 3 = Bit 3

The buffer can store up to ten readings. Each call to this function returns the next oldest result. The buffer is circular, so the results will repeat after 10 function calls.

Reading of switches and storing of results occurs, whether the USB board is powered or not.

8.6.1 Function Prototype

BOOL NextSecuritySwitchRead(LPDWORD time, LPBYTE switches);

8.6.2 Programming Considerations

The SECURITY pipe is used for this function.

8.7 Function name: StartSecuritySwitchRead ()

Calling this function ensures that the next NextSecuritySwitchRead() call will return the most recent result in the buffer. See NextSecuritySwitchRead() for more details.

8.7.1 Function Prototype

BOOL StartSecuritySwitchRead(void);

8.7.2 Programming Considerations

The SECURITY pipe is used for this function.

8.8 Function name: ClearSecuritySwitches()

Calling this function will clear the security switch buffer. See the NextSecuritySwitchRead() function for more details.

8.8.1 Function Prototype

BOOL ClearSecuritySwitches(void);

8.8.2 Programming Considerations

The SECURITY pipe is used for this function.

8.9 Function name: ReadAndResetSecuritySwitchFlags()

This function returns the security switch flags that have opened and closed since the last time the function was called. The security flags are subsequently reset.

ReadAndResetSecuritySwitchFlags( ) is a slow Security Pipe function. A fast version is available called CachedReadAndResetSecuritySwitchFlags( ), and this function is recommended when real time performance is important in your code.

8.9.1 Function Prototype

```
BOOL ReadAndResetSecuritySwitchFlags( LPBYTE closedSwitches, LPBYTE openSwitches );
```

8.9.2 Programming Considerations

The SECURITY pipe is used for this function.

Either ReadAndResetSecuritySwitchFlags( ) or CachedReadAndResetSecuritySwitchFlags( ) should be used in your code. They should not be used interchangeably.

8.10 Function name: CachedReadAndResetSecuritySwitchFlags()

This function is a fast version of ReadAndResetSecuritySwitchFlags( ), and is recommended if real time performance is important in your code. The functionality is slightly different in that the previous instead of the current security switch status is returned. If a switch change occurs, then two calls to CachedReadAndResetSecuritySwitchFlags( ) would be required before the change is visible.

The majority of X10 API calls work at relatively high speeds, limited mainly by USB transfer constraints (4ms per message). Security Pipe functions work at slow speeds because the onboard PIC performs the processing for these operations via a slow I^2C bus.

When the function ReadAndResetSecuritySwitchFlags( ) is called, the following events occur:

1. A USB packet is sent to the X10 requesting the security switch flags.
2. The X10 asks the PIC chip for the current security switches (this is where most time is taken).
3. The X10 returns the PIC response to the PC via the USB channel.

CachedReadAndResetSecuritySwitchFlags( ) changes the above sequence as follows:

1. A USB packet is sent to the X10 requesting the security switch flags.
2. The X10 returns the previously obtained security switch flags to the PC via the USB channel.
3. The X10 communicates with the PIC chip requesting the current security switch states.

Stages 2 and 3 are reversed and the previous security switches are immediately returned to the PC. If the function is called again before stage 3 is complete then the response will be delayed until stage 3 is complete from the previous call. It is therefore recommended that a minimum of a one second gap should occur between calls to CachedReadAndResetSecuritySwitchFlags( ).

8.10.1 Function Prototype

```
BOOL CachedReadAndResetSecuritySwitchFlags(     LPBYTE closedSwitches,
                                                LPBYTE openSwitches );
```

8.10.2 Programming Considerations

The SECURITY pipe is used for this function.

Either ReadAndResetSecuritySwitchFlags( ) or CachedReadAndResetSecuritySwitchFlags( ) should be used in your code. They should not be used interchangeably.

8.11 Function Name: EnableRandomNumberGenerator()

This function will enable the random number generator in the X10. Following this, a new number will be available to read every second. Input 2 can be polled to check for a new random number. When a new number is available IP2 will be read high, subsequent reads of IP2 will be read low, until the next number is ready.

This means that input 2 cannot be used as a normal input while the random number generator is enabled. IP 2 can be polled using either GetInputs(), GetChangedInputs() or GetRawInputs() functions.

The <seed> parameter is a 32-bit value that will seed the random number sequence; it should therefore be a different value each time this function is executed. This function can be called at any time without calling the corresponding disable function, allowing the random number sequence to be reseeded.

A good way to ensure randomness is to reseed the random number sequence with the current time when the player of a game performs a certain “random” action, for example adds more credits, rather than simply when the game first loads up.

8.11.1 Function Prototype

BOOL EnableRandomNumberGenerator(DWORD seed);

8.11.2 Programming Considerations

The SECURITY pipe is used for this function, but the PIC is not involved, therefore there is no I^2C overhead.

8.12 Function Name: DisableRandomNumberGenerator()

This function will disable the random number generator in the 8051 firmware, allowing input 2 to be used as a normal input.

8.12.1 Function Prototype

BOOL DisableRandomNumberGenerator()

8.12.2 Programming Considerations

The SECURITY pipe is used for this function, but the PIC is not involved, therefore there is no I^2C overhead.

8.13 Function Name: GetRandomNumber()

This function is used to fetch the current random number available from the random number generator in the 8051 firmware.

8.13.1 Function Prototype

BOOL GetRandomNumber(LPWORD randnumber);

8.13.2 Programming Considerations

The SECURITY pipe is used for this function, but the PIC is not involved, therefore there is no I^2C overhead.

9 CCTALK

The term “cctalk” is the protocol name for a single wire bi-directional serial interface using NRZ data transmission. It is used to communicate with coin acceptors and hoppers. It is a multi-drop protocol with one master device polling a number of slaves. Transmission rates are normally 9600 baud.

Interface signal voltage levels are 0V and either +5V or +12V. Care must be taken when connecting a number of devices to one port: if they use a mixture of +5V and +12V levels, they may not be compatible with each other. In this case, the +5V device *may* prevent the +12V device from seeing a proper high (+12V) level.

Serial port A or serial port B (or both) can be configured for cctalk operation. There are two modes of cctalk operation available: Mode 0 and Mode 1.

9.1 Mode 0

This mode of operation should be used when the PC-based software controls all cctalk messages.

The functions that are available are:

- Send data: BOOL Send(usbSerialPort port, LPBYTE data, UINT length);
- Read receive buffer: BOOL Receive(usbSerialPort port, LPBYTE data, LPUINT length);
- Read receive buffer with time stamping: BOOL ReceiveByteWithTimestamp(usbSerialPort port,LPBYTE dst,LPBYTE interval,BOOL *bReceived);
- Set a time out message: BOOL SetTimeoutMessage(usbSerialPort port, LPBYTE message, BYTE numBytes, float timeout);

The send message is “blocking” and the Windows Driver will not return to the calling program until all of the data has been transmitted.

The receive-with-time-stamp function enables the PC-based program to check the timing of the received data. This can be useful because cctalk message do not have a unique “start of message” or “end of message” character. If message synchronisation is lost, the only way of identifying the start of a new message is that there will be a gap of more then 10ms since the last character.

The ability to set a time-out message has been added so that the Firefly X10 can disable a cctalk device, such as a coin acceptor, if the PC-based software appears to have stopped running.

9.2 Mode 1

cctalk mode 1 has been added for situations in which the Firefly X10 needs to provide greater assistance to the PC-based software.

The purpose of Mode 1 is to allow the Firefly X10 to send regular messages (“polls”) to a number of devices and to store the response. For each port (A or B), a number of devices can be defined: each one is given a number of parameters to define how it should be polled. These include:

- The message to be sent
- How often it should be sent
- How long to wait for a reply

cctalk bus received data includes “local echo” i.e. it includes the transmitted data. In Mode 1, the Firefly X10 will *remove* local echo characters and it will also *remove* responses to poll messages that are identical to the last stored (but not yet read) response. Once the response has been read and deleted, then the next response will be always be stored.

For example, the following sequence will occur:

1. <poll_reply_1>: stored
2. <poll_reply_1>: repeat, ignored
3. <poll_reply_2>: stored
4. <poll_reply_3>: stored
5. <poll_reply_3>: repeat, ignored

If the PC reads the buffer, it will read <poll_reply_1>. If it processes and deletes this message, then the next buffer read will give <poll_reply_2>. Similarly, if it processes and deletes this message, then the next buffer read will give <poll_reply_3>.

Since the buffer is now *empty*, the next poll will store the reply even if it is <poll_reply_3> *again*. Thus, the next buffer read will give <poll_reply_3> after the next automatic poll.

The reason for using a separate function call to delete the message from the buffer is that this allows the PC to process the message without the risk of losing it if power is lost. (The buffer in the Firefly X10 is battery-backed and will not be lost if power is removed). The PC should not delete the message until it has processed it and updated any data as a result. Typically, this would involve updating a coin counter, possibly also in battery-backed SRAM on Firefly X10, in response to a poll message that indicates that a coin has been received.

It is also possible to define a message that should be sent only once. This gives the PC-based software a mechanism for sending individual message to individual devices. If there are three or fewer cctalk devices on a port, then it is simplest to allocate a spare device to single messages so that polling does not need to be stopped and re-started. For example, if there is a coin acceptor and payout hopper on Port A, then devices 0 and 1 are used to poll the coin acceptor and payout hopper respectively, but device 2 is used to send and receive single messages to either device whilst polling continues.

NOTE: the cctalk Mode 0 commands are *NOT* available in Mode 1.

10 USING REELS

The X10 is capable of driving up to three reels simultaneously. There is, however, no flexibility in allocation of particular outputs to particular reels: if one reel is configured, it will use outputs OP16/17/18/19 and input IP6. The second reel, if configured, will use outputs OP20/21/22/23 and input IP7. The third reel, if configured, will use outputs OP24/25/26/27 and input IP8.

The coils are centre-tapped and the outputs are arranged so that the first pair of outputs are for coil A and the second pair are for coil B. Taking reel 1 as an example, outputs OP16/17 are for coil A and outputs OP18/19 are for coil B.

The input detects the vane as a high pulse on a normally low digital signal. Note – if the signal from the stepper reel is opposite to this, then the X10 will still drive the reel, but will report position errors when running backwards. In this situation, the position errors should be ignored.

The correct procedure for initialising reels is as follows:

1. Use the ConfigureReelsEx( ) function. This configures the number of active reels - valid values are 0 to disable, 1, 2 and 3. This function also configures the number of half-steps per complete reel turn and the number of steps per symbol – please note that these values are shared between all configured reels.
2. Call the function SpinRampUp( ) to define the ramp up table. This function must be called for each active reel.
3. Call the function SpinRampDown( ) to define the ramp down table. This function must be called for each active reel.
4. Call the function SetDutyCycle( ) to define the reel resting duty cycle. This function must be called for each active reel.
5. Call the function ReelSynchroniseEnable( ) and then spin the reel for a complete turn using the function SpinReels( ). This step is required to synchronise the reel position counter, and must be followed for each active reel.

Once the above steps are completed then the reels are correctly configured. It is now possible to spin reels using the SpinReels( ) function in addition to obtaining reel position information using the GetReelStatusEx( ) function.

11 USING SECURITY SWITCHES

This section details the mechanism by which the X10 deals with power off security switches SW1, SW2, SW3 and SW4.

Every security switch change is logged, subject to a maximum buffer size of ten entries, along with the corresponding time at which the change occurred. These events are stored in a circular buffer in the X10 security chip.

It is envisaged that the X10 security switch log will be accessed as follows:

1. Call GetClock() to read the time. If this is wrong, then the battery has been tampered with or the security chip has been removed and refitted. This means that the security switch readings have been lost. Otherwise, if the clock is correct, the security switch readings are also correct.
2. Call function StartSecuritySwitchRead() in order to begin reading the log.
3. Call NextSecuritySwitchRead() as many times as necessary to read all of the results.
4. Call ClearSecuritySwitches() if you wish to clear the buffer. New results will be added to the buffer whether you decide to call this function or not. If you do not call this function, your software will need to remember which switch buffer results have already been processed.

11.1 Obtaining current security switch flags

The buffer can only log ten events. If more than ten events have occurred then the oldest results will be lost. This may be deliberate – someone might operate one security switch several times to hide a different switch result by pushing it out of the buffer. This condition can be checked by calling the function ReadAndResetSecuritySwitchFlags(): this will tell you if other security switches have changed, even if they are no longer in the buffer because it has overflowed.

This function can also be used to “peek” the switches. Call ReadAndResetSecuritySwitchFlags() twice. The first call resets the flags. The second call will then tell you which switches are now open and which are now closed.

ReadAndResetSecuritySwitchFlags() is a slow function because it needs to communicate with the onboard PIC via a slow I^{2}C link. The function will “hold” until the PIC communications have completed. A fast version is also available called CachedReadAndResetSecuritySwitchFlags(). This function is faster because it immediately returns the result of the previous read and triggers a new read, ready for the next function call. So if a switch change has occurred, two calls would be required before it is visible, and three calls would be required to “peek” the switches.

The following table demonstrates the different behaviour of ReadAndResetSecuritySwitchFlags() and CachedReadAndResetSecuritySwitchFlags():

Switch Pos	**ReadAndResetSecuritySwitchFlags** *Closed : Open : Changed*	**CachedReadAndResetSecuritySwitchFlags** *Closed : Open : Changed*
0000	1111 : 0000 : 0000	1111 : 0000 : 0000
0000	1111 : 0000 : 0000	1111 : 0000 : 0000
0000	1111 : 0000 : 0000	1111 : 0000 : 0000
1111	1111 : 1111 : 1111	1111 : 0000 : 0000
1111	0000 : 1111 : 0000	1111 : 1111 : 1111
1111	0000 : 1111 : 0000	0000 : 1111 : 0000