X-line X10 / X10i / X15 - Software User Manual

Document No. 80-18374 Issue 14   HEBER LTD

Current Issue: -   Issue 14 – 23rd October 2014
Previous Issues: -   Issue 13 – 3rd June 2014
                          Issue 12 – 11th April 2011

HEBER

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

HEBER

CONTENTS

HEBER

HEBER

This page intentionally left blank.

HEBER

1   INTRODUCTION

This document describes all aspects of the X-line X10/X10i/X15 development API. In addition to detailing every available API function this document also discusses subjects which must be taken into account when designing your software (for example ccTalk, speed constraints, security etc).

This software user guide is divided into the following chapters:

- **Chapter 1 – Introduction:** This chapter.
- **Chapter 2 – Processing Speed Considerations:** Discusses some board limitations which must be taken into account when designing your software. Also discusses the differences between X10, X10i and X15 performance.
- **Chapter 3 – The X10i Security Library:** Covers the X10/X10i security features.
- **Chapter 4 – X15 Security:** Covers the X15 security features.
- **Chapter 5 – System Functions:** Details the API specifically concerned with system functions.
- **Chapter 6 – IO Pipe Functions:** Covers the API specifically concerned with IO functions.
- **Chapter 7 – Serial Pipe A and Serial Pipe B Functions:** Covers the API specifically concerned with serial communications.
- **Chapter 8 – SPI Pipe Functions:** Covers the API specifically concerned with SPI communications.
- **Chapter 9 – Memory Pipe Functions:** Covers the API specifically concerned with EEPROM storage.
- **Chapter 10 – SRAM Pipe Functions:** Covers the API specifically concerned with SRAM storage.
- **Chapter 11 – Security Pipe Functions:** Covers the API specifically concerned with security.
- **Chapter 12 – C Interface** Discusses possibilities of application programming in the 'c' language.
- **Chapter 13 – CCTALK:** Discusses various aspects of ccTalk programming.
- **Chapter 14 – Using Reels:** Discusses various aspects of driving reels.
- **Chapter 15 – Using Security Switches:** Describes the methods envisaged for reading security switches.
- **Chapter 16 – API Return Codes:** Details all X10/X10i/X15 API return codes.

Note: Throughout this manual references that are made to the "X10i" product also apply equally to the "X10" and "X15" products unless otherwise noted. The X10i and X15 products offer all of the X10 functionality as well as some additional or alternate functions.

Return Values: With very few exceptions each of the API functions returns a BOOL value indicating the exit status of the function; a TRUE return indicates success, a FALSE value indicates failure. The precise cause of a failure can be obtained by a call to GetLastError() function and referring to chapter-16. Where the return value is other than BOOL this is noted in the text.

NOTE:

From October 2014 the XLine support software has been enhanced so that either 32-bit or 64-bit XLine applications can be run on 64-bit (Windows or Linux) platforms. The installation of a 64-bit development kit now automatically installs both the 32-bit and 64-bit development kit components. To help avoid confusion, 64-bit related files have '_x64' added to the filenames. The Linux 'file' utility and its Windows equivalent can also be used to identify whether an executable files or DLL/.so file is compiled for a 32-bit or 64-bit platform.

Downloaded 32-bit development kits continue to operate, as previously, on 32-bit (Windows or Linux) platforms only.

2 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%. 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% of the 1ms interrupt is used. If all 3 reels are configured, 29% 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 13% 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 13 for ccTalk mode descriptions), 5% of the interrupt is used. So if all 4 ccTalk devices are configured on port A then 19% of the interrupt will be consumed, and if all 4 devices on port B are also used then this value will double to 38 % of the interrupt.

2.1.5 SPI

During an SPI transaction, the 1ms interrupt increases by 10%.

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^{2}C 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.

2.3 X10 vs X10i Performance Comparison

The X10i has been designed to offer performance at least equal to that of the X10. In addition, it has some useful feature additions that are documented throughout this manual.

The 8051 processor on the X10i is faster than the one on the X10 so the 1ms interrupt load on the X10i is generally lower than on the X10.

The X10i has two modes of operation: full speed (when connected to a USB 1.1 hub) and high speed (when connected to a USB 2 hub). When running in full speed the performance is similar between the X10 and the X10i, however it is possible for the X10i performance to occasionally drop slightly when many pipes are utilised simultaneously. When running in high speed the X10i performance is generally higher particularly during large data transfers (e.g. SRAM reads/writes).

2.4 X10i vs X15 Product Comparison

The X15 is based on the core design of the X10i product using the same 8051 processor but with a number of feature improvements and additions to the X10i design. These features include:

- Enhanced security using 3-DES encryption
- Updated and improved secure PIC
- Addition of a backplane connector (compatible with the X20 product)
- Additional 512k RAM storage.

3   THE X10I SECURITY LIBRARY

The X10i provides advanced security features that prevent other people running your game code. This section outlines the procedure for utilising these security features.

When an X10i is first plugged into a PC it will initialise into a "locked" state. It cannot be successfully used until it is "unlocked". This is performed by the "UnlockIO" library which must be linked with your program. The header file "unlockio.h" must also be included at the beginning of your program.

Supplied with the X10i Development Kit is a generic UnlockIO library and a generic PIC. When the generic UnlockIO library is linked into your program and a call to UnlockX10() is made then the X10i should unlock and full functionality will be enabled.

The generic UnlockIO library is useful for game development, however before a game goes into production it is recommended that you obtain a "secure" UnlockIO library and PIC. Heber provides this secure combination with the guarantee that each is unique. When the game is linked against a secure library then it will not run on an X10i board which is not fitted with the corresponding secure PIC just as an X10i with a secure PIC will not operate unless it is unlocked with the corresponding library. In order to use a secure PIC you must replace the generic UnlockIO library with the secure UnlockIO library and re-build the software.

The RelockHardware() call can be used to relock the X10i hardware after unlockX10() call has succeeded. Once RelockHardware has been called all unlockX10 calls will silently fail for the next 60 seconds. See the security pipe section for further details.

The UnlockIO library provides three functions that are each discussed in the following paragraphs.

3.1   Function name: UnlockX10()

This function unlocks the X10i. It returns TRUE if no problems were encountered during the unlock process and FALSE otherwise. If the X10i was previously unlocked then this function will simply return and not attempt another unlock.

Function Prototype
```
BOOL UnlockX10( FireFlyUSB * FireFly );
```


3.2   Function name: UnlockX10Recheck()

This function is provided as an alternative to UnlockX10(). The UnlockX10Recheck() function differs from UnlockX10() in that it always attempts to unlock the X10i. If the X10i was previously unlocked and UnlockX10Recheck() is called (using the incorrect *unlockio* library) then the unlock process will fail and the X10i will re-lock itself.

Function Prototype
```
BOOL UnlockX10Recheck( FireFlyUSB * FireFly );
```


3.3   Function name: VerifyX10UnlockLibrary()

This function ensures that the *unlockio* library is correctly linked to the fitted X10i. It does not attempt to unlock or re-lock the X10i. The function returns TRUE if the *unlockio* library matches and FALSE otherwise.

Function Prototype
```
BOOL VerifyX10UnlockLibrary( FireFlyUSB * FireFly );
```

4   X15 SECURITY

4.1   Overview

The X15 security mechanism consists of three hardware elements plus security software components built in to the X-line libraries. The hardware components are:

- A trusted customer-specific hardware Smartcard (supplied by Heber).
- A trusted customer-specific secured PIC device (supplied by Heber).
- An un-trusted PC upon which the game runs.

The mechanism includes a security watchdog with an initial timeout of four minutes from the moment the board is powered up, or the Init() function is invoked. Before this four minute timeout elapses, the application software needs to initialize the security system and perform a first security Authentication process which if successful will reset the watchdog timeout to four minutes. Authentication requests must be continually issued at intervals thereafter to prevent the watchdog timer from expiring (every 1.5 minutes is recommended). If the application fails to perform the authentication process or the authentication should fail (eg. because Smartcard secured PIC or software libraries are not a matched set), all I/O activity of the X15 board is disabled when the watchdog times out. The only way to re-enable X15 I/O at that point is to perform a hardware reset.

Software initialization of the security system requires that functions SmartcardReset() and SmartcardSetSessionID(), along with the first Authentication() request, be issued within the first four minutes after power-up, and that subsequent Authenticate requests occur at lease every 4 minutes or less.

4.2   Function name: SmartcardReset()

This function must be called within four minutes of power-up or the issuing of an Init() function call to prepare the Smartcard for operation. This function will be followed by a call to function SmartcardSetSessionID() to fully initialise the security system ready for authentication cycles.

The pointer 'rxMessage' identifies a buffer space used to receive the card Answer-To_Reset message. A 64 byte buffer should be supplied.

'rxLength' will reflect the actual length of the Answer-To-Reset message returned in buffer rxMessage.

Parameter 'msDelay' should be set to 150.

Function Prototype
```
BOOL SmartcardReset    LPBYTE rxMessage, unsigned int *rxLength, UINT msDelay );
```

4.3   Function name: SmartcardSetSessionID()

A standard part of Smartcard communication is to define a SessionID to be used in all subsequent card communications. An arbitrary value can be supplied.

The retuned value of 'sw1sw2' is again part of standard Smartcard communication. The returned value can be ignored.

The final parameter 'msDelay' should be set to 600.

Function Prototype
```
BOOL SmartcardSetSessionID   ( UINT sessionID, unsigned int *sw1sw2, UINT msDelay );
```

4.4 Function name: Authenticate()

The authenticate() function needs to be continually called, at intervals, after power-up to prevent the X15 I/O lines from being disabled. Heber recommends the function be called approximately every 1.5 minutes. Perhaps the simplest way of accomplishing this is to place the authentication process in a separate thread from the main application software.

Due to the communications involved, the Authenticate command takes some time to complete. It is best to allow at least one second for this.

The first function parameter is a pointer to a 192-bit encrypted key as supplied to the customer by Heber.

The sessionID, as specified in the call to SmartcardSetSessionID() is included as the second parameter.

For security reasons, the Authenticate() function does not report if the authentication process was successful or not (if, for example, the wrong encrypted key was used). However, errors are reported if there were communication problems preventing the authentication from completing (e.g. if a Smartcard is not inserted, a Smartcard read error occurs or a PIC write error occurs)

Function Prototype

```
bool Authenticate( unsigned char *encryptedKey, unsigned int sessionID );
```

4.5 X15 Security Example

The following is a snippet of code demonstrating security system initialization and one authentication cycle:

```
unsigned char EKey[] = { 0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef,
                         0xed,0xcb,0xa9,0x87,0x65,0x43,0x21,0x00,
                         0x01,0x23,0x45,0x67,0x89,0xab,0xcd,0xef );

int main( void )
{
    fireflyUSB X15;
    unsigned int sessionID, sw1sw2, rxLength;
    unsigned char *rxReset;

    printf( "Establishing link with X15 device..." );
    if ( !X15.Init( ) )
    {
        printf( "error - %d.\n\n", (int)X15.GetLastError( ) );
        return ( 1 );
    }
    printf( "success.\n\n" );

    if ( !X15.SmartcardReset( rxReset, &rxLength, 150 ) )
    {
        printf( "Reset error-%d.\n", (int)X15.GetLastError( ) );
        return ( 1 );
    }
    printf( "Setting session ID..." );

    sessionID = 0x123456;
    if ( !X15.SmartcardSetSessionID( sessionID, &sw1sw2, 300 ) )
    {
        printf( "error - %d.\n\n", (int)X15.GetLastError( ) );
        return ( 1 );
    }
    printf( "success.\n\n" );

    printf( "Performing the first authentication cycle..." );
    if ( !X15.Authenticate( Ekey, sessionID ) )
    {
        printf( "error - %d.\n\n", (int)X15.GetLastError( ) );
        return ( 1 );
```

```
        }
        printf( "success.\n\n" );

        return( 0 );
}
```

5 SYSTEM FUNCTIONS

5.1 Function name: FireFlyUSB()

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

Function Prototype
```
FireFlyUSB()
```

Programming Considerations
None. No pipes are used for this function.

5.2 Function name: FireFlyUSB(FireFlyUSB &)

A library supplied copy constructor for FireFlyUSB objects. Provides proper copy construction behaviour, and allows temporary FireFlyUSB objects to be passed to functions by value or reference.

Function Prototype
```
FireFlyUSB( FireFlyUSB &)
```

Programming Considerations
None. No pipes are used for this function.

5.3 Function name: operator = (FireFlyUSB &)

Supports assignment of FireFlyUSB objects.

Function Prototype
```
FireFlyUSB & operator = ( FireFlyUSB &)
```

Programming Considerations
None. No pipes are used for this function.

5.4 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.

Function Prototype
```
~FireFlyUSB( );
```

Programming Considerations
None. No pipes are used for this function.

5.5 Function name: InitUSBBoard( )

Initialises an X-line board. 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 X-line 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.

Function Prototype
```
BOOL InitUSBBoard( BYTE boardNumber );
```

Programming Considerations
None. No pipes are used for this function.

5.6 Function name: init( )

Initialises the only X-line USB device. This is a simplified version of the previous function. It can be used when there is only one X-line board connected to the system and the address switch setting is to be ignored.

Function Prototype
```
BOOL init( );
```

Programming Considerations
Do not use this function if more than one X-line board is connected to the system. No pipes are used for this function.

5.7 Function name: close( )

Closes the USB device.

Function Prototype
```
BOOL close( );
```

Programming Considerations
No pipes are used for this function.

5.8 Function name: GetFittedBoard()

This function is ascertains which board is currently fitted.

Function Prototype
```
BOOL GetFittedBoard( LPBYTE fittedBoard );
```

Programming Considerations
The fittedBoard parameter is a pointer to a BYTE which will contain information about the currently fitted board.

Four values can be returned:
- UNIDENTIFIED_BOARD : The currently fitted board is unknown
- X10_BOARD : An X10 is fitted
- X10I_BOARD : An X10i is fitted
- X15_BOARD : An X15 is fitted

5.9 Function name: GetBoardSpeed()

The X10i/X15 boards can work at two different speeds: if the board is plugged into a USB 1.1 hub then it will work at "full speed", and if it is plugged into a USB 2 hub then it will work at "high speed". High speed is considerably faster than full speed and this allows many X10i functions to perform faster.

The X10 board can only work in "full speed" mode. Plugging it into a USB 1.1 or a USB 2 hub makes no difference.

This function returns the speed at which the currently fitted board is working at.

Function Prototype
```
BOOL GetBoardSpeed( LPBYTE boardSpeed );
```

Programming Considerations
The boardSpeed parameter is a pointer to a BYTE which reports the speed at which the currently fitted board is working at.

Three possible values can be returned:

- UNKNOWN_SPEED : The speed is unknown so therefore the board is not behaving correctly.
- USB_1_1_FULL_SPEED : The board is working at full speed.
- USB_2_0_HIGH_SPEED : The board is working at high speed. The X10 cannot return this value.

5.10 Function name: GetProductVersion( )

This function reports the X-line Development Kit Product version.

Function Prototype
```
BOOL GetProductVersion( LPBYTE versionProduct );
```

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

No pipes are used for this function.

5.11 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.

Under Linux this function will report the version of the 'fflyusb.so' shared library.

Function Prototype
```
BOOL GetDllVersion( LPBYTE versionDll );
```

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

No pipes are used for this function.

5.12 Function name: Get8051Version()

This function reports the version number of the 8051 software. This is downloaded to the X-line device during enumeration.

Function Prototype
```
BOOL Get8051Version( LPBYTE version8051 );
```

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

The Memory pipe is used for this function.

5.13 Function name: GetLastError( )

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

Function Prototype
```
usbErrorCode GetLastError( );
```

Return value
A variable of type usbErrorCode that describes the reason for failure, also see chapter 16 on *API Return Codes*.

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.

6 IO PIPE FUNCTIONS

6.1 Function name: GetInputBit()

Note: This function is now deprecated and is supported only for compatibility.

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.

Function Prototype

```
BOOL GetInputBit( usbInputBitId input_bit_id, LPBOOL result );
```

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.

As this function is provided for compatibility only, for new designs use the function GetInputs(usbInput * usbInputs) which returns the state of all inputs in a data structure.

6.2 Function name: GetInputs(Byte Pointer)

Note: This function is now deprecated and is supported only for compatibility reasons.

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)

Function Prototype

```
BOOL GetInputs( LPBYTE inputs );
```

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.

As this function is provided for compatibility only, for new designs use the function GetInputs(usbInput * usbInputs) which returns the state of all inputs in a data structure.

6.3 Function name: GetInputs(Structure Pointer)

All of the inputs on the USB board may be read into a data structure using this function call. The structure of type usbInput is defined in x10idefs.h as:

```
typedef struct
{
    BYTE      byIn[3];          /* Inputs [0..2] */
    BYTE      bySw;             /* Sw input      */
    BYTE      byCs;             /* Cs input      */
} usbInput;
```

Function Prototype

```
BOOL GetInputs( usbInput *inputs );
```

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. This function should be used instead of deprecated function GetInputs(Byte Pointer).

6.4 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.

Function Prototype

```
BOOL SetOutputBit( usbOutputBitId output_bit_id, BOOL bit_state );
```

Programming Considerations

The outputs are updated once every millisecond.

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

As this function is provided for compatibility only, for new designs the functions SetOnPeriodOutputBit and the related functions (described below) are recommended.

6.5 Function name: SetOutputs()

Note: This function is now deprecated and is supported only for compatibility reasons.

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.

Function Prototype

```
BOOL SetOutputs( LPBYTE outputs );
```

Programming Considerations

The outputs are updated once every millisecond.

This function actually uses function ModifyOutputs(), so a direct call to ModifyOutputs() will be slightly more efficient.

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

As this function is provided for compatibility only, for new designs the function SetOnPeriodOutputs( ) and the related functions (described below) are recommended.

6.6 Function name: GetChangedInputs()

Reports the inputs that have changed since the last input read. 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.

Function Prototype

```
BOOL GetChangedInputs( usbInput *changedInputs );
```

6.7 Function name: ConfigureChangedInputsCallback()

This function allows the user to specify what changes in input signals to the X-line board should invoke a user-supplied Callback function to process those changed inputs. The process has been implemented to work with debounced inputs only (to filter out switch noise etc.).

The user supplies pointers to:
- a usbInput data structure called 'changedInputsMask'. Specify a '1' for any input position where any transition (low-to-high or high' to 'low) is if interest.
- a usbInput data structure called 'changedInputsLevel'. Specify a '1' for each input position whose transitions from low-to-high is if interest; similarly specify a '0' for bit positions where transitions from high-to-low are of interest.
- address of the user callback function which is to be invoked when the input conditions specified above are satisfied.

When invoked, the callback function delivers pointers to two usbInput structures:
- the first structure reports '1's' identifying the input(s) that satisfied the conditions to invoke the callback.
- the second reports the current state of all input lines (equivalent to issuing GetInputs()).

The prototype for the callback function is:
```
void callback_routine(usbInput *changedInputs, usbInput *currentInputs )
```

Only one set of input conditions and one callback function can be armed at any time. Subsequent calls to ConfigureChangedInputsCallback() will disarm any existing trigger conditions and re-arm with the new conditions and callback function. To disable an armed set of callback conditions entirely invoke the ConfigureChangedInputsCallback() function with the callback function address set to NULL.

While the callback mechanism is armed and in use, the API functions GetChangedInputs() and GetInputs() should be avoided because of the risk of contention; the GetInputs() function may read and reset input signal changes before the callback mechanism gets a chance to detect the signal change.

Function Prototype

```
BOOL ConfigureChangedInputsCallback ( usbInput *changedInputsMask,
                                       usbInput *changedInputsLevel,
                                       void (*InputsCallbackFunction)(
                                               usbInput *changedInputs,
                                               usbInput *currentInputs )
                                     );
```

6.8 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 x10idefs.h. When using this function the inputs are not de-bounced first; the instantaneous input values are returned.

Function Prototype

```
BOOL GetRawInputs( usbInput *rawInputs );
```

6.9 Function name: InputMultiplexing()

This function is used to either enable or disable input multiplexing. Once enabled then outputs OP12, OP13, OP14 and OP15 are used exclusively 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.

Function Prototype

```
BOOL InputMultiplexing( usbInputMultiplexing input  );
```

6.10 Function name: GetMultiplexedInputs()

The use of multiplexed input lines is outlined in the following diagram. Note that blocking diodes must be fitted to each switch-line used to prevent channels from interfering with one another.
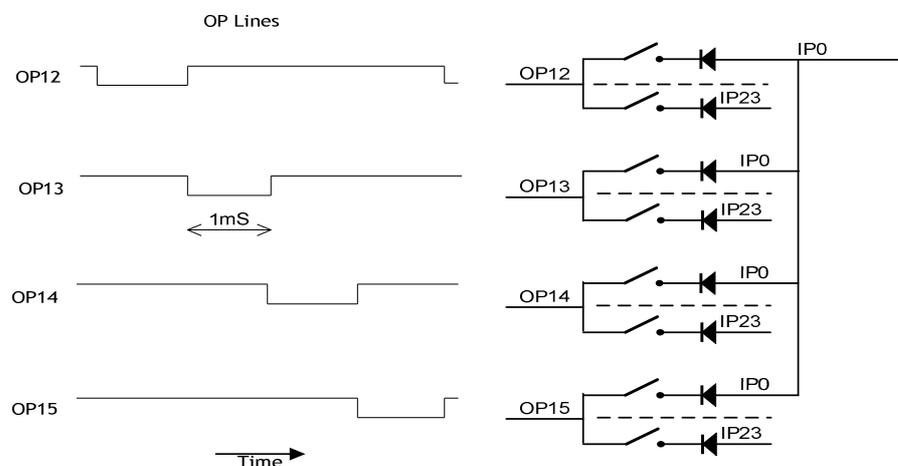
The function GetMultiplexedInputs returns the state of the multiplexed inputs. The data is returned as the following structure (which is defined in 'x10io.h'):

```
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 all 24 inputs IP0-IP23 (24 bits = 3 bytes).

There are 4 channels and each channel is controlled by one of output pins 12, 13, 14 or 15 as shown in the diagram. For the first 1mS time slice, output 12 is toggled low while outputs 13, 14, and 15 are set high. Input lines IP0-IP23 are read and stored as channel 0. Next output 12 is toggled high and output 13 is toggled low and the 24 input bits stored as channel 1. This procedure is repeated until all input channel have been read, as summarised in the table below. Effectively each individual multiplexed input channel is read once every 4mS.

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

Function Prototype
```
BOOL GetMultiplexedInputs( usbMultiplexedInput *inputs );
```

6.11 Function name: ModifyOutputs()

Note: This function is now deprecated and is supported only for compatibility reasons.

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);
```

This 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.

As this function is provided for compatibility only, for new designs the functions SetOnPeriodOutputBit and the related functions (described below) are recommended.

Function Prototype

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

6.12 Function name: SetOnPeriodOutputs()

Historically functions SetOutputs, SetOutputBit and ModifyOutputs were used to manipulate the Output lines OP0-OP31 and AUX0-5 lines. For new designs Output bit manipulation is best performed using SetOnPeriodOutput and related functions. These allow a PWM (pulse width modulation) dimming mechanism to be employed to each Output line.

Operation of the Output dimming mechanism is described in the following paragraphs.

The overall pulse-width time-window for the dimming mechanism is 10mS, consisting of 10 one-millisecond intervals. This 10mS window is divided into two sub-periods:
- the on-period.
- the off-period.

The duration of the on-period is defined by the SetBrightness function and can range in value from 0-10. For example if SetBrightness is set to 7, then the on-period will be the first 7ms of each 10mS window, the remaining 3mS will be the off-period. Brightness-0 is the same as 'off' all the time, and brightness-10 the same as 'on' all the time.

Function SetOnPeriodOutputs defines what the states of all output lines (OP0-OP31 and AUX0-5) should be during the 'on'-period part of the cycle.

Similarly function SetOffPeriodOutputs defines what the states of all output lines (OP0-OP31 and AUX0-5) should be during the 'off'-period part of the cycle.

Hence the collection of all outputs (OP0-OP31 and AUX0-5) alternate between two sets of defined states during every 10mS period.

Four possibilities emerge from this.

Take OP0 as an example output, and assume SetBrightness has been set to 7:
- if we were to set OP0 to '1' in both SetOnPeriodOutputs and SetOffPeriodOutputs functions, the output will be 'on' for both on and off parts of cycle, hence will be 'on' all the time.
- If we were to set OP0 to '0' in both SetOnPeriodOutputs / SetOffPeriodOutputs functions, output OP0 will be 'off' all the time.
- if we set OP0 to '1' with SetOnPeriodOutputs, and to '0' with SetOffPeriodOutputs, OP0 will be 'on' for the first (7mS) part of the 10mS window, and 'off' for the remaining (3mS) of the cycle, hence the output will appear slightly dimmed compared to an output that is 'on' all the time.
- finally we could set OP0 to '0' with SetOnPeriodOutputs, and to '1' with SetOffPeriodOutputs so output OP0 would be 'off' for the first 7mS part of the cycle, and 'on' for the remaining 3mS part of the cycle, output OP0 will appear more dimmed than the previous case.

By the choice of SetPrightness we can in effect arrange for 4 brightness levels to be selectable at any time: output 'off', output 'on' but very dimmed, output brighter but below full brightness, and finally output at full brightness.

The functions SetOnPeriodOutputs and SetOffPeriodOutputs allow you to set all outputs OP0-OP31 and AUX0-5 (5 bytes) in a single function call; the alternative functions SetOnPeriodOutputBit and SetOffPeriodOutputBit allow modification of individual output or auxilary bits, one-bit for each time the function is called.

Operation of the dimming mechanism will be overridden by use of PulseOutput or PusleOffOutput on the specific output line being pulsed for the pulse duration specified.

Function Prototype

```
BOOL SetOnPeriodOutputs( usbOutput onPeriods );
```

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.

6.13 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.

Function Prototype

```
BOOL SetOffPeriodOutputs( usbOutput offPeriods );
```

Programming Considerations

The outputs are updated once every millisecond.

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

6.14 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.

Function Prototype

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

Programming Considerations

The outputs are updated once every millisecond.

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

6.15 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.

Function Prototype

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

Programming Considerations

The outputs are updated once every millisecond.

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

6.16 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.

Function Prototype

```
BOOL SetOutputBrightness( BYTE brightness );
```


6.17 Function name: PulseOutput()

This function is used to pulse an X-line output and hence determine whether a meter current is detected.

The parameter <outputNumber> is the output to be pulsed. This value corresponds to X-line 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.

Using this function will override the SetOnPeriodOutputs and related functions dimming control for the duration of <pulseDurationMs>.

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

Function Prototype

```
BOOL PulseOutput( BYTE outputNumber, BYTE pulseDurationMs );
```


6.18 Function name: PulseOffOutput()

This function is used to pulse an X-line 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 output to be pulsed. This value corresponds to X-line 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.

Using this function will override the SetOnPeriodOutputs and related functions dimming control for the duration of <pulseDurationMs>.

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

Function Prototype

```
BOOL PulseOffOutput( BYTE outputNumber, BYTE pulseDurationMs );
```

6.19 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.

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

Programming Considerations
Reliable current detection needs the presence of an external 12V power supply and will not work if the X-line's only power source is USB bus power. The absence of an external 12V power supply can be inferred if the current sense input is true and all the parallel outputs are off.

6.20 Function name: ConfigurePulsedInput()

This function has been deprecated. You should use *ConfigurePulsedInputEx()* instead.

This function is used to configure reading of input pulses on parallel devices, for example coin acceptors. The following functions 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.

Function Prototype
```
BOOL ConfigurePulsedInputEx(   usbInputBitId inputBitID,
                        BYTE pulseLowerTime,
                        ActiveState inputActiveState );
```

Programming Considerations
This function has been deprecated. You should use *ConfigurePulsedInputEx()* instead.

6.21 Function name: ConfigurePulsedInputEx()

This function is used to configure reading of input pulses on parallel devices, for example coin acceptors. The following functions 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 <pulseUpperTime> specifies the maximum pulse length, in ms, for valid input pulses. A value of 0 will disable maximum pulse length checking. If any input pulses exceed this value then the input will become "jammed" and further pulses will not be recorded. See function GetPulsedInputStatus() to obtain the jammed status of an input.

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

Function Prototype

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

Programming Considerations

This function replaces the deprecated function *ConfigurePulsedInput()*.

This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

6.22 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.

Function Prototype

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

6.23 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.

Function Prototype

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

6.24 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.

Function Prototype

```
BOOL ResetPulsedInputCounter( usbInputBitId inputBitID );
```

6.25 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.

Function Prototype
```
BOOL DecrementPulsedInputCounter( usbInputBitId inputBitID );
```

6.26 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.

Function Prototype
```
BOOL ReadPulsedInputCounter( usbInputBitId inputBitID, BYTE *inputCounterValue);
```

6.27 Function name: GetPulsedInputStatus()

This function reports the "Jammed" status of a parallel input. An input becomes Jammed if the pulse has remained active for more than it's upper limit (as defined in ConfigurePulsedInputEx()). Once an input is Jammed then further pulses (whether valid or not) will not be recorded.

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

The pointer parameter <jamStatus> reports TRUE if the input is jammed and FALSE otherwise.

The pointer parameter <coinsReleased> reports the number of coins that have currently been released.

Function Prototype
```
BOOL GetPulsedInputStatus( usbInputBitId inputBit,
                           BOOL *jamStatus,
                           BYTE *inputCounterValue );
```

6.28 Function name: ReleaseParallelHopperCoins()

This function has been deprecated. The function *ReleaseParallelHopperCoinsEx()* should be used instead.

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 calls to *GetParallelHopperStatus()* should 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 begins counting pulses – these pulses equate to released coins. If each coin is not released within a specified timeout time, it is assumed that an error has occurred. In this event the motor 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 the input is active state. This can be one of the following four values (two on an X10):

- Low – the input is active low. Complete low pulses (with a minimum length specified in pulseLowerTime) are identified as released coins. Immediately following the final coin pulse, the motor is turned off.
- High – the input is active high. Complete high pulses (with a minimum length specified in pulseLowerTime) are identified as released coins. Immediately following the final coin pulse, the motor is turned off.

- FallingEdge – the input is active low. Low pulses (with a minimum length specified in pulseLowerTime) are identified as released coins. After a delay of pulseLowerLength into the final coin pulse, the motor is turned off. This is useful for very sensitive hoppers because the board does not wait for the complete final pulse before turning the motor off. This is only available on an X10i or X15.

- RisingEdge – the input is active high. High pulses (with a minimum length specified in pulseLowerTime) are identified as released coins. After a delay of pulseLowerLength into the final coin pulse, the motor is turned off. This is useful for very sensitive hoppers because the board does not wait for the complete final pulse before turning the motor off. This is only available on an X10i or X15.

The parameter <outputBit> specifies the 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 each 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.

The counting input specified for counting coins during the Release command remains active after the Release command. Call EndPulsedInputCheck() to disable this counter.

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

Programming Considerations
This function has been deprecated. You should use *ReleaseParallelHopperCoinsEx()* instead.

6.29 Function name: ReleaseParallelHopperCoinsEx()

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 calls to *GetParallelHopperStatus()* should 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 any of the required coins are not released within a specified timeout it is 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 <pulseUpperTime> specifies the maximum pulse length, in ms, for valid input pulses. A value of zero implies an infinite maximum pulse length (equivalent to calling ReleaseParallelHopperCoins() function).

Note: If the maximum pulse width is exceeded then the motor will be turned off and the status (when calling *GetParallelHopperStatus()*) will be set to *Jam*.

The parameter <inputActiveState> specifies the input is active state. This can be one of the following four values:

- Low – the input is active low. Complete low pulses (with a minimum length specified in pulseLowerTime and maximum length specified in pulseUpperTime) are identified as released coins. Immediately following the final coin pulse, the motor is turned off.
- High – the input is active high. Complete high pulses (with a minimum length specified in pulseLowerTime and maximum length specified in pulseUpperTime) are identified as released coins. Immediately following the final coin pulse, the motor is turned off.

- FallingEdge – the input is active low. Low pulses (with a minimum length specified in pulseLowerTime and maximum length specified in pulseUpperTime) are identified as released coins. After a delay of pulseLowerLength into the final coin pulse, the motor is turned off. This is useful for very sensitive hoppers because the board does not wait for the complete final pulse before turning the motor off.

- RisingEdge – the input is active high. High pulses (with a minimum length specified in pulseLowerTime and maximum length specified in pulseUpperTime) are identified as released coins. After a delay of pulseLowerLength into the final coin pulse, the motor is turned off. This is useful for very sensitive hoppers because the board does not wait for the complete final pulse before turning the motor off.

The parameter <outputBit> specifies the 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 each 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.

The counting input specified for counting coins during the Release command remains active after the Release command. Call EndPulsedInputCheck() to disable this counter.

Function Prototype

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

Programming Considerations

This function replaces the deprecated function *ReleaseParallelHopperCoins()*.

If a further call is made to ReleaseParallelHopperCoins() before a previous dispense is allowed to complete the error status USB_MESSAGE_COIN_HOPPER_BUSY is returned.

If when a call to ReleaseParallelHopperCoins() is issued the counting input pin is already active, the error status USB_MESSAGE_COIN_HOPPER_INIT_JAM is reported and the motor does not start. *ReleaseHopperCoinsStatus* will also report InitJam.

This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

6.30 Function name: StopHopperCoinRelease()

This function will stop the release of coins from a parallel hopper by turning off the motor. The motor is stopped immediately, without any check whether a coin is already part way through the hopper mechanism.

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

Function Prototype
```
BOOL StopHopperCoinRelease( usbInputBitId inputBit );
```

Programming Considerations
This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

6.31 Function name: StopHopperCoinReleaseSafe()

This function differs from StopHopperCoinRelease() in that it keeps the motor running until dispensing of the current coin is complete and before starting the dispense of the next coin. This should avoid a possible 'Jam' condition that may be caused by stopping the hopper motor immediately in the middle of a dispensing sequence.

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

Function Prototype
```
BOOL StopHopperCoinReleaseSafe( usbInputBitId inputBit );
```

Programming Considerations
This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

6.32 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.

The parameter <status> is a pointer to a structure of type *ReleaseHopperCoinsStatus*. This is continuously updated during coin release. It will be set to one of six values:
- *Running* – A request to dispense coins has been received, the hopper motor has started, and coins are being released.
- *Success* – The requested number coins have been successfully released.
- *Failure* - The coin timeout has expired before all coins were released. *coinsReleased* reports how many coins were actually released.
- *Stopping* (only available on the X10i and X15) – This status will be reported for a brief time from when the command *StopHopperCoinReleaseSafe* is issued until the hopper motor has reached the correct stopping position.
- *InitJam* (only available on the X10i and X15) – A request to dispense coins found the count input line already active before starting the next dispense, indicating an already jammed mechanism.
- *Jam* (only available on the X10i and X15) - The time taken to release a coin exceeded the maximum pulse length, thus indicating a jam.

The parameter <coinsReleased> is a pointer to a BYTE which will contain the number of coins that were actually dispensed by the last coin released command.

Function Prototype
```
BOOL GetParallelHopperStatus(  usbInputBitId inputBit,
                               ReleaseHopperCoinsStatus *status,
```

```
BYTE *coinsReleased );
```

6.33 Function name: ConfigureReels()

This function must be called if reels are to be controlled by the X-line 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:* 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: 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.

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

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.

6.34 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().

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

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

6.35 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. 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.

Function Prototype

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

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.

6.36 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). 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.

Function Prototype

```
BOOL SpinRampUp( BYTE reelNumber, usbRampTable rampUpTable );
```

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.

6.37 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). 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.

Function Prototype

```
BOOL SpinRampDown( BYTE reelNumber, usbRampTable rampDownTable );
```

6.38 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.

Function Prototype

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

Programming Considerations

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

6.39 Function name: GetReelStatus()

Note: This function is now obsolete. The Function **GetReelStatusEx()** (documented in the next section) 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().

Function Prototype

```
BOOL GetReelStatus( usbReelStatus * status );
```

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.

Note: On a reels initial spin the position field data is invalid until the X-line board has detected the presence of the opto flag and the synchronised field has been set.

6.40 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().

Note: On a reels initial spin the position field data is invalid until the X10i has detected the presence of the opto flag and the synchronised field has been set.

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.

Function Prototype

```
BOOL GetReelStatusEx( ReelStatusEx * status );
```

Programming Considerations

Use this function regularly to check that reels remain synchronised.

6.41 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().

Function Prototype

```
BOOL ReelSynchroniseEnable( BYTE reelNumber );
```

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: Reels will only synchronise when the reel spins past the position-0 opto-detector while spinning **forwards** (positive direction/step size value).

7 SERIAL PIPE A AND SERIAL PIPE B FUNCTIONS

7.1 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 is not used. Parity checking is automatically enabled if *Parity* (see below) is set to ODDPARITY or EVENPARITY.

Parity sets the parity method. This can be set to the following values: NOPARITY (no parity), ODDPARITY (odd parity), EVENPARITY (even parity) or DATA_BIT_9PARITY (parity bit is used for a ninth data bit). If this member is ODDPARITY or EVENPARITY, parity checking is performed and errors are reported using the function *GetParityErrors()*.

Note: DATA_BIT_9PARITY is a special case. In this case, the ninth bit to be transmitted or received is an additional data bit, not a parity bit. This mode can be set if <type> is set to PORT_RS232 or PORT_CCTALK, but ***not if one of the polled modes is set***. This is because 9-bit parity requires the use of word-wide buffers instead of byte-wide buffers: polled modes do not support word-wide buffers.

Note: To send and receive 9 bit serial data, set *Parity* = DATA_BIT_9PARITY and use the following functions: Send9BitData(), Receive9BitData() or ReceiveByteWithTimestamp9BitData(). Do not use any other serial functions. Do not use these 9 bit functions with any other value of *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 can be 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).

Subject to configuration, serial operation may need the presence of an external 12V power supply and may not work if the only source of X-line board power is the USB bus power. The absence of an external 12V power supply can be inferred if the current sense input is true and all the parallel outputs are off.

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

7.2 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.

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

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

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

7.3 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.

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

Programming Considerations
The X-line board stores received serial data in a 600-byte circular buffer. If more than 600 bytes have been received since the last call to Receive() then the oldest data will be lost.

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

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

7.4 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.

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

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.

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

7.5 Function name: Send9BitData()

This function is the same as Send() above, but must be used if 9-bit serial transmission is required. SetConfig() must have set *Parity* to DATA_BIT_9PARITY for this function to work correctly.

Function Prototype
```
BOOL Send9BitData ( usbSerialPort port, LPWORD data, UINT length );
```

Programming Considerations
See Send() above.

7.6 Function name: Receive9BitData()

This function is the same as Receive() above, but must be used if 9-bit serial reception is required. SetConfig() must have set *Parity* to DATA_BIT_9PARITY for this function to work correctly.

Function Prototype
```
BOOL Receive9BitData ( usbSerialPort port, LPWORD data, LPUINT length );
```

Programming Considerations
See Receive() above.

7.7 Function name: ReceiveByteWithTimestamp9BitData()

This function is the same as ReceiveByteWithTimestamp() above, but must be used if 9-bit serial reception is required. SetConfig() must have set *Parity* to DATA_BIT_9PARITY for this function to work correctly.

Function Prototype
```
BOOL ReceiveByteWithTimestamp9BitData (    usbSerialPort port,
                                           LPWORD rxWord,
                                           LPBYTE interval,
                                           BOOL *received );
```

Programming Considerations
See ReceiveByteWithTimestamp() above.

7.8 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.

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

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

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

7.9 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.

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

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

7.10 Function name: SetSASMachineAddress()

This function sets the machine address for SAS (Slot Accounting System) operation.

The 'address' parameter specifies the 7-bit machine address. If this is set to zero then SAS functionality is disabled.

The SAS algorithm can be summarised as follows:

- If SAS Address set (using this function) then if *any* address word is received (bit 9 set) then stop any transmit, if one is in progress
- If the polled address was *ours*, start an auto reply timer, which until expired will define a "window of opportunity" for the host PC to generate its own response
- Prepare an automated response, but don't send it yet:
    - If bit 8 of address was set then prepare a normal poll response transmit_9bit_serial(0x000)
    - If bit 8 of address was clear then prepare a long poll response transmit_9bit_serial(SASMachineAddress,0x000)
- If the host PC supplies us with transmit data before the auto reply timer has expired, then send it and cancel the auto reply. Make a note that message transmission was successful
- If, on the otherhand, the PC fails to ask us to transmit data in time, then send the prepared response when the timer expires. Make a note that message transmission failed.

- Once the timer has expired, prevent further transmissions requested by the PC until a new poll of our machine has been received and the timer has been restarted. If a message must be sent without a poll (a "chirp"), then *this can only be done by disabling auto reply first*.

If serial data is in the process of transmission using Send9BitData() and an address word is received then transmission will cease and Send9BitData() will return an error code USB_MESSAGE_SAS_ADDRESS_INTERRUPT.

Use the GetSASMessageStatus() function to find out if the host PC responded to the poll in time.

Function Prototype

```
BOOL SetSASMachineAddress( usbSerialPort port, BYTE address );
```

Programming Considerations

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

7.11 Function name: SetSASAutoReply()

This function sets the SAS auto reply enable/disable and timing. See previous subsection (Function name: SetSASAddress()) for a description of SAS operation.

The 'AutoReplyEnabled' parameter specifies whether auto reply is enabled or not; either TRUE (enabled) or FALSE (disabled).

The 'AutoReplyDelayMs' parameter specifies the delay in milliseconds (minimum 0, maximum 255) before the automatic reply is sent.

Function Prototype

```
BOOL SetSASAutoReply (   usbSerialPort port,
                         BOOL AutoReplyEnabled,
                         BYTE AutoReplyDelayMs );
```

Programming Considerations

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

If 'AutoReplyEnabled' is TRUE, it is _not_ possible to transmit outside the period of 'AutoReplyDelayMs' milliseconds following a poll of this machine's SAS address.

7.12 Function name: SetSASBusy()

This function is identical in operation to SetSASAutoReply() above, except that the auto reply delay is fixed at 0 milliseconds. By setting the 'busyState' to TRUE, the host PC is indicating to the X-line board that the host PC is too busy to respond and the X-line board can respond to any polls immediately without waiting to see if the host PC prepares a response in time.

The 'busyState' parameter specifies the SAS busy state; either TRUE (auto reply enabled) or FALSE (not enabled).

Equivalent to BOOL SetSASAutoReply (usbSerialPort port, BOOL busyState, 0);

Function Prototype

```
BOOL SetSASBusy( usbSerialPort port, BOOL busyState );
```

Programming Considerations

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

7.13 Function name: GetSASMessageStatus()

This function reports whether the response to the last poll of this machine's address was supplied by the host PC within the required time window (TRUE), or an automated reply had to be sent because the auto reply timer expired (FALSE).

The 'messageSuccess' parameter will be set TRUE (PC responded) or FALSE (automated response went).

Function Prototype

```
BOOL GetSASMessageStatus ( usbSerialPort port, BOOL *messageSuccess );
```

Programming Considerations

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

7.14 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 an X-line board: 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 selecting 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 involved in polling. The *Once* option is used to send a single message one time only to a specific device (see Chapter-13). 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 triggered 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 X-line board copy of 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 X-line board 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. *Note:* 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. *Note:* 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 <min_buffer_space>. 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. After 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: Step-1 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 stop polling. If the port is configured to the *Once* option, that port, and any 'next_trigger' devices chained from it, will poll just once.

Function Prototype
```
BOOL ConfigureCCTalkPort( usbSerialPort port, CCTalkConfig *cctalkConfig );
```

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.

Page 42

This function is only available in ccTalk Mode 1.

7.15 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 an X-line board: 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.

Document No. 80-18374 Issue 14

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. 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.

Function Prototype
```
BOOL ConfigureRS232Poll( usbSerialPort port, RS232PollConfig *pollConfig );
```

Programming Considerations
The function SetConfig() must be called first to ensure that the serial port is correctly configured and set to PORT_RS232_POLLED operation.

This function is only available in RS232 Polled Mode.

7.16 Function name: SetPolledHostTimeout()

This function sets a host activity timeout. If the host software has not requested serial data (using ReceivePolledMessage()) for the defined timeout then the X-line board will inhibit the device. This function is useful for disabling money acceptors in the event of the host PC application crashing.

Three 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.

The <timeout> parameter specifies the host timeout in seconds. It can take the value 0 – 25.5, where zero disables the timeout. The timeout is disabled by default.

Function Prototype
```
BOOL SetPolledHostTimeout(      usbSerialPort port,
                                BYTE deviceNumber,
                                double timeout);
```

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.

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

7.17 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.

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

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.

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

7.18 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.

Function Prototype

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

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.

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

7.19 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.

Function Prototype

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

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.

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

8   SPI PIPE FUNCTIONS

8.1   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 X-line 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 X-line input IP18 is also used as a data input from the SPI device.

Function Prototype
```
BOOL EnableSPI( void );
```

8.2   Function name: DisableSPI()

This function disables the SPI protocol. This releases the X-line SPI outputs OP0 and OP1 for use by other functions.

Function Prototype
```
BOOL DisableSPI( void );
```

8.3   Function name: SendSPI()

This function provides communications to a SPI device. 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.

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

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

There is a size limit of 20 bytes (160 bits) on both transmit and receive messages.

8.4   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. This will contain the complete received message including checksum etc.

Function Prototype

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

Programming Considerations

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

There is a size limit of 16 bytes for both transmit and receive messages.

9 MEMORY PIPE FUNCTIONS

9.1 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 X-line board initialisation in an attempt to detect and configure the currently fitted EEPROM.

Function Prototype
```
BOOL CheckEEPROM( X10EEPROM *eepromConfig );
```

9.2 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 X-line board initialisation in an attempt to detect and configure the currently fitted EEPROM.

Function Prototype
```
BOOL ConfigureEEPROM( X10EEPROM *eepromConfig );
```

9.3 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.

Function Prototype

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

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.

9.4 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.

Function Prototype

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

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.

10 SRAM PIPE FUNCTIONS

10.1 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.

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

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.

10.2 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.

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

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.

10.3 Function name: ReadLargeSRAM()

Function Prototype
```
BOOL ReadLargeSRAM( DWORD address, LPBYTE data, DWORD totalLength );
```

Programming Considerations
This function is very similar to ReadSRAM(), however it is possible to address up to 0x70000 on the X10i instead of 0x7e00 and up to 0xF0000 on the X15. This means that the function can address up to 448k of SRAM on the X10i and 960k on the X15 instead of 32k (as on the X10).

This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

10.4 Function name: WriteLargeSRAM()

Function Prototype

```
BOOL WriteLargeSRAM( DWORD address, LPBYTE data, DWORD totalLength );
```

Programming Considerations

This function is very similar to WriteSRAM(), however it is possible to address up to 0x70000 on the X10i instead of 0x7e00 and up to 0xF0000 on the X15. This means that the function can address up to 448k of SRAM on the X10i and 960k on the X15 instead of 32k (as on the X10).

This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

11 SECURITY PIPE FUNCTIONS

11.1 Function name: GetPICVersion()

This function reports the version of X-line PIC (security device) firmware.

Function Prototype
```
BOOL GetPICVersion( LPBYTE versionPIC );
```

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

11.2 Function name: GetPICSerialNumber()

This function reports the serial number of the PIC (which is blank when supplied by Heber). If the serial number is not set by the user (using SetPICSerialNumber) then 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.

Function Prototype
```
BOOL GetPICSerialNumber( LPBYTE serialNumberPIC );
```

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

11.3 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.

Function Prototype
```
BOOL SetPICSerialNumber( LPBYTE serialNumberPIC );
```

11.4 Function name: GetDallasSerialNumber()

This function reports the serial number of the Dallas chip on the X-line board. Every board has a unique serial number. The first byte returned is the family code of 0x01. The next 6 bytes are a unique serial number and the final 8th byte is a CRC byte. Although the CRC byte is made available to the user, the data has already been tested for integrity using the CRC and the result is returned in crcValid.

GetDallasSerialNumber() is a slow Security Pipe function.

Function Prototype
```
BOOL FireFlyUSB::GetDallasSerialNumber(    LPBYTE serialNumberDallas,
                                           LPBYTE crcValid )
```

Programming Considerations
This returns a string of 8 bytes in serialNumberDallas. It sets crcValid = TRUE if the data passes the 8-bit CRC test.

This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

11.5 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.

Function Prototype
```
BOOL SetClock( DWORD time );
```

11.6 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.

Function Prototype
```
BOOL GetClock( LPDWORD time );
```

11.7 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.

Function Prototype
```
BOOL NextSecuritySwitchRead( LPDWORD time, LPBYTE switches );
```

11.8 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.

Function Prototype
```
BOOL StartSecuritySwitchRead( void );
```

11.9 Function name: ClearSecuritySwitches()

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

Function Prototype
```
BOOL ClearSecuritySwitches( void );
```

11.10 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.

Function Prototype

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

Programming Considerations

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

11.11 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 X-line 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^{2}C bus.

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

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

Function Prototype

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

Programming Considerations

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

11.12 Function name: ReadAndResetBatteryFailFlag()

This function is used to find out if the battery voltage for the battery-backed SRAM and PIC fell below the minimum threshold whilst powered-down. It sets batteryFlag TRUE if the battery voltage had fallen too low. It also clears the flag so that it will not return TRUE again until the battery voltage has fallen too low again.

ReadAndResetBatteryFlag( ) is a slow Security Pipe function.

Function Prototype

```
BOOL ReadAndResetBatteryFailFlag( LPBYTE batteryFlag );
```

Programming Considerations

This function is only supported on the X10i and X15 boards. It is not supported on the X10 board.

11.13 Function Name: EnableRandomNumberGenerator()

This function will enable the random number generator in the X-line board. 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.

Function Prototype

```
BOOL EnableRandomNumberGenerator( DWORD seed );
```

Programming Considerations

The PIC is not involved so there is no I^2C overhead.

11.14 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.

Function Prototype

```
BOOL DisableRandomNumberGenerator()
```

Programming Considerations

The PIC is not involved so there is no I^2C overhead.

11.15 Function Name: GetRandomNumber()

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

Function Prototype

```
BOOL GetRandomNumber( LPWORD randnumber );
```

Programming Considerations

The PIC is not involved so there is no I^{2}C overhead.

11.16 Function Name: RelockHardware()

This function causes the X-line security lock to be reactivated. To unlock the board after this call an application needs to wait at least 60 seconds and then call the unlockio library.

Function Prototype

```
BOOL RelockHardware( void );
```

Programming Considerations

This function is only supported on the X10 and X10i boards. It is not supported on the X15 board.

11.17 Function name: ReadAndResetRTCFailure()

This function is used to find out if a failure has occurred with the Real Time Clock on the PIC at any time, the *rtcflag* returns TRUE if a failure has been detected. After performing this call the flag is reset. Calling this function multiple time can be used to determine if a prolonged failure has occurred.

ReadAndResetRTCFailure( ) is a slow Security Pipe function.

Function Prototype

```
BOOL ReadAndResetRTCFailure( unsigned char* rtcFlag );
```

Programming Considerations

This function is only supported on the X15 board. It is not supported on the X10 and X10i boards.

11.18 Function name: ReadClockAtBatteryFailure()

This function is used to determine the time read at the moment at which the battery voltage for the battery-backed SRAM and PIC fell below the minimum threshold whilst powered-down.

ReadClockAtBatteryFailure( ) is a slow Security Pipe function.

Function Prototype

```
BOOL ReadClockAyBatteryFailure ( LPDWORD time );
```

Programming Considerations

This should be used in conjunction with the function ReadAndResetBatteryFlag( ).
The function is only supported on the X15 board. It is not supported on the X10 and X10i boards.

11.19 Function name: SmartcardReset()

Resets the Smart Card. This function should be called at the beginning of your program.

The <rxMessage> parameter is a pointer to a buffer when the Smart Card return message is stored.

The <rxLength> parameter will specify the length of rxMessage.

The <msDelay> parameter specifies the delay, in ms, between the Smart Card write and the Smart Card read. Depending on the command issued, the Smart Card takes a specific amount of time to complete the task. For this task, a value of 150ms is recommended.

Function Prototype
```
bool SmartcardReset( LPBYTE *rxMessage,
                     unsigned int *rxLength,
                     unsigned int msDelay );
```

11.20 Function name: SmartcardSetSessionID()

Sets the Smart Card session ID. This function must be called prior to attempting an authentication.

The <sessionID> parameter is a 32-bit integer containing the session ID. This value must be passed to the Authenticate function.

The <sw1sw2> parameter is a pointer to a word where the Smart Card status is returned.

The <msDelay> parameter specifies the delay, in ms, between the Smart Card write and the Smart Card read. Depending on the command issued, the Smart Card takes a specific amount of time to complete the task. For this task, a value of 300ms is recommended.

Function Prototype
```
bool SmartcardSetSessionID( unsigned int sessionID,
                            unsigned int *sw1sw2,
                            unsigned int msDelay );
```

11.21 Function name: SmartcardGetApplicationID()

Returns an ASCII string containing the Smart Card application ID.

The <sw1sw2> parameter is a pointer to a word where the Smart Card status is returned.

The <idString> parameter is a pointer to a buffer where the ID string will be stored. The ID string is a maximum of 60 characters.

The <msDelay> parameter specifies the delay, in ms, between the Smart Card write and the Smart Card read. Depending on the command issued, the Smart Card takes a specific amount of time to complete the task. For this task, a value of 300ms is recommended.

Function Prototype
```
bool SmartcardGetApplicationID( unsigned int *sw1sw2,
                                unsigned char *idString,
                                unsigned int msDelay );
```

11.22 Function name: Authenticate()

Authenticates the X15. Read the chapter '*X15 Security*' for more information on authentication.

In order to prepare the Smart Card for subsequent authentication cycles, a call to *SmartcardReset()* and *SmartcardSetSessionID()* should be made during your game initialisation.

The <encryptedKey> parameter points to a buffer containing the 192-bit (24-byte) encrypted key as supplied by Heber.

The <sessionID> parameter is the 32-bit Smart Card session ID. This must be the same as the value previously passed to *SmartcardSetSessionID()*.

Function Prototype

```
bool Authenticate( unsigned char *encryptedKey,
                   unsigned int sessionID );
```

12 C-PROGRAMMING INTERFACE

The API functions described in this manual have all been implementation in the C++ Language. For those who wish to use the C-language instead, a collection of C wrapper functions and libraries are also available. For further details please contact: Support@heber.co.uk.

13 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, each capable of controlling up to 4 devices. There are two modes of ccTalk operation available: Mode 0 and Mode 1.

13.1 Mode 0

This mode of operation should be used where all ccTalk communication is to be directly controlled by the application software.

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 application program to check the timing of the received data. This can be useful because ccTalk messages 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.

A time-out message facility is also available, so that the X-line board can automatically disable a ccTalk device, such as a coin acceptor, if communication with the application software is lost for any reason.

13.2 Mode 1 (in effect CCTALK_POLLED_MODE)

ccTalk mode 1 is intended for situations where the X-line board itself handles a great deal of the routine device polling and status gathering, relieving the application program of the software overhead necessary performing those routine tasks..

Mode 1 allows the X-line board to send regular messages ("polls") to a range of ccTalk devices and to store the various responses in reply. Each physical port (port-A and port-B) can be associated (by device address) with up to four ccTalk devices: each of the four device associations (or *channels*) is configured with a number of parameters to define which ccTalk device it should poll and how it should be polled. These include:

- The ccTalk device to be addressed
- The poll message to be sent
- How often it should be sent
- How long to wait for a reply

Data received over the ccTalk bus will automatically include "local echo" (i.e. will include a copy of the transmitted data). In Mode 1, the X-line board automatically *remove* local echo characters. The x-line board will also *remove* responses to poll messages that are identical to the previous response message stored (but not yet

read by the application). After the response message has been read and deleted by the application software, then any subsequent response will always be stored.

An example of processing a sequence of reply messages might be as follows:

1. <poll_reply_1>: stored
2. <poll_reply_1>: because it is a repeat, it will not stored
3. <poll_reply_2>: stored
4. <poll_reply_3>: stored
5. <poll_reply_3>: again is a repeat so is not stored

Suppose the application reads the buffer, it will read <poll_reply_1>. If the application processes and deletes this message, then the next buffer read operation will report <poll_reply_2>. Again if the application processes and deletes this message, then the next buffer read will report <poll_reply_3>.

Because the reply buffer is now empty, the next reply received will be automatically stored, even if that reply happens to be a repeat of <poll_reply_3>, hence the next buffer read will report <poll_reply_3>.

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 X-line board is battery-backed and the buffer contents will not be lost if power is removed). The PC should not delete the message until it has processed it and updated any dependent data variables. For example, the response to a poll message that indicates that a coin has been received might involve updating a coin counter variable (possibly also in battery-backed SRAM on the X-line board).

It is also possible to define a message that should be sent only once. This gives the application software a mechanism for sending individual message to individual devices. Suppose there are three or less ccTalk devices assigned to automatic polling via a port, it is simplest to allocate the remaining spare channel to single messages so that automatic polling does not need to be stopped and re-started. For example, if there is a coin acceptor and payout hopper on Port A allocated to channel-0 and 1, these might be combined into an automatic poll sequence; channel-2 could then be used to send and receive single messages to either device, without disturbing automatic polling of coin acceptor or the payout hopper.

Note: The ccTalk Mode 0 commands are *NOT* available in Mode 1.

13.3 CCTALK Polling Operations

All <parameter> values referred to in the following descriptions are parameters supplied with the ConfigureCCTALKPort() function. The word channel describes the association between one of the four associations . The association between one of the x-line ccTalk the ccTalk device

The polling cycle for each CCTALK device operates as follows.
- Each device initially waits through the <polling_interval> as specified when the device was configured.
- After its <polling_interval> elapses, the device sends the <poll_msg> programmed for that device to its attached cctalk peripheral.
- Waits for up to <max_response_time> to receive a reply from the device. If a reply is not received, the <inhibit_msg> is sent to the device and the device is inhibited. Otherwise the device is ready to begin another polling cycle

So in summary the cycle consists of: Wait -- Poll -- Reply.

To manage this polling process, the x-line board repeatedly checks all configured cctalk devices (at 10mS intervals) to determine whether each device at this moment is (a) disabled, (b) waiting through <polling_interval>, (c) is ready to transmit <poll_message> or (d) is waiting through <maxResponseTime> for a cctalk reply packet. The x-line board also manages the serialisation of polling requests where more than one poll request is due at any instant in time.

Whether poll cycles as described above occur just once or in a continually repeating sequence, depends on the parameter <method>. If <method> is set to 'Repeated' then the polling cycle is continuous, repeating every

<polling_interval> until either (a) the device fails to respond or (b) the <method> is changed to specify 'Disabled'.

When automatic repeated polling is not required and just a single message needs to be sent to a device the <method> is set to 'Once'. If a delay is required before the single message is sent, the <polling_interval> is set as needed.

Lastly, where we wish to chain a group of devices together so that all of the group perform their respective polls one immediately after the other, then the method <Triggered> is used. One device will need to act as the start of the chain with its <method> set to 'Once' or 'Repeated'. The parameter <next_trigger_device> of that first device points to next device in the chain. This second device will have its <method> set to 'Triggered'. If a further device is to be included in the chain then <next_trigger_device> again points to the following device, and once again will need to have its <method> set to 'Triggered'. The end of the device chain will be the device with its <next_trigger_device> set to 'NO_TRIGGER'.

The polling operations of RS232 polled mode behave in a very similar way to that described above. The key difference is that RS232 polling is configured using the ConfigureRS232Poll() function which supplies some extra configuration parameters compared to CCTALK polling. See section 7.15 for further details.

14 USING REELS

The X-line board 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. If the signal from the stepper reel is opposite to this, then the board 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 reel symbol or graphic – 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 at least 1 complete revolution using the function SpinReels( ). This step is required to synchronise the reel index position and counter, and must be issued 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.

15 USING SECURITY SWITCHES

This section details the mechanism by which the X-line board 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 X-line board security chip.

It is envisaged that the 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.

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^2C 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

16 API RETURN CODES

The following table describes the X-line API return codes (as obtained using GetLastError()). The codes (along with their corresponding numerical values) are defined in x10idefs.h.

Code	Return Code	Meaning
0	USB_MESSAGE_EXECUTION_SUCCESS	No error – successful operation.
1	USB_MESSAGE_NOT_RECOGNISED	The API is sending an unrecognised command to the board. This can be because the API/drivers do not match the firmware on your X-line board.
2	USB_MESSAGE_LENGTH_ERROR	The API is sending an incorrect packet size to the board. This can be because the API/drivers do not match the firmware on your X-line board.
3	USB_MESSAGE_EXECUTION_FAILURE	An X-line command failed.
4	USB_DEVICE_WRITE_ERROR	The API is unable to write over USB link. Try reconnecting the board.
5	USB_DEVICE_READ_ERROR	The API is unable to read over USB link. Try reconnecting the board.
6	USB_DEVICE_RESPONSE_ERROR	The API obtained an invalid response from the X-line board. This can be because the API/drivers do not match the firmware on your X-line board.
7	USB_DEVICE_BYTE_COUNT_ERROR	The API is receiving an incorrect packet size from the X-line board. This can be because the API/drivers do not match the firmware on your board.
8	USB_MESSAGE_PARAMETER_OUT_OF_RANGE	An argument passed to the API is invalid.
9	USB_MESSAGE_UNKNOWN_HANDSHAKE_MODE	Invalid serial handshake mode specified.
10	USB_MESSAGE_ILLEGAL_BAUD_RATE	Invalid serial baud rate specified.
11	USB_MESSAGE_UNKNOWN_SERIAL_PROTOCOL	Invalid serial protocol specified.
12	USB_MESSAGE_I2C_NACK	An I^{2}C 'nack' occurred when communicating with the PIC.
13	USB_MESSAGE_I2C_ERROR	An I^{2}C error occurred when communicating with the PIC.
14	USB_MESSAGE_EXECUTION_TIMEOUT	The X-line command timed out.
15	USB_MESSAGE_SRAM_ADDRESSING_ERROR	An invalid address has been specified when reading/writing SRAM.
16	USB_MESSAGE_REEL_NUMBER_ERROR	An invalid reel has been specified.
17	USB_MESSAGE_REEL_BUSY	The specified reel is currently busy.
18	USB_MESSAGE_SPIN_TABLE_LENGTH_ERROR	An invalid number of entries have been specified in the spin table.
19	USB_MESSAGE_TOO_FEW_REEL_STEPS	Not enough spin steps have been specified. The minimum number of steps is: number of steps in up ramp + number of steps in down ramp - 1
20	USB_MESSAGE_REEL_STATUS_LENGTH_ERROR	The API requested an invalid number of bytes from the X-line board. This can be because the API/drivers do not match the firmware on your board.
21	USB_MESSAGE_SYNC_REQUEST_LENGTH_ERROR	The API requested an invalid number of bytes. This can be because the API/drivers do not match the firmware on your board.
22	USB_MESSAGE_PIPE_UNAVAILABLE	An attempt to use an invalid X-line pipe has been made. This can be because the API/drivers do not match the firmware on your board.

23	USB_MESSAGE_RANDOM_GENERATOR_DISABLED	An attempt to obtain a random number has been made when the generator is disabled.
24	USB_MESSAGE_PIPE_CURRENTLY_USED	An attempt has been made to use the same X-line pipe by two simultaneous threads/processes.
25	USB_MESSAGE_FUNCTION_NOT_SUPPORTED_ON_X10	An attempt has been made to call a function not supported on the X10.
26	USB_MESSAGE_FUNCTION_NOT_SUPPORTED_ON_X10i	An attempt has been made to call a function not supported on the X10i.
27	USB_MESSAGE_SAS_ADDRESS_INTERRUPT	An SAS interrupt has occurred.
28	USB_MESSAGE_TRY_AGAIN	A Windows-specific USB transaction failure occurred. This has been fixed in the latest drivers.
29	USB_MESSAGE_PARALLEL_INPUT_CURRENTLY_ACTIVE	An attempt has been made to release hopper coins when an input is still active. The input must be inactive before releasing coins.
30	USB_MESSAGE_ILLEGAL_POINTER	A NULL pointer was passed to an API function.
31	USB_MESSAGE_FUNCTION_NOT_SUPPORTED_ON_X15	An attempt has been made to call a function not supported on the X15.
32	USB_MESSAGE_SMARTCARD_SW1SW2_ERROR	An error has occurred while writing or reading information to or from the Smartcard.
33	USB_MESSGAE_PIC_BUSY	A call made to a function within the PIC could not be executed as the PIC is busy executing a previous call.
34	USB_MESSAGE_PIC_BUFFER_FULL	A buffer on the PIC has become full due to the PIC not having sufficient time to process the data in the buffer prior to subsequent calls adding further data to the buffer causing it to become full.
37	USB_MESSAGE_THREAD_FAIL	ChangedInputsCallback thread mechanism failure.
38	USB_MESSAGE_FUNCTION_NOT_CONFIGURED	An attempt has been made to use part of the API that has not been configured. For example, you must call SetConfig() prior to calling any of the other serial port functions.
39	USB_MESSAGE_COIN_HOPPER_BUSY	An attempt has been made to issue a second hopper coin dispense before previous dispense completed.