"Naples" PC Lamp/Reel/Switch Controller Applications Interface User Manual

Table of Contents

Revision History

Version	Date	Author	Description
0.0 - Draft	24-Aug-06	D. Bush A. Graham	Initial description document.
0.1 - Draft	26-Sep-06	D. Bush	Lamp intensity changed.
1.2	5-Dec-06	A. Graham	Added description of VFD interface. Updated description of the ramp descriptor tables to reflect longer acceleration and deceleration tables.

Introduction

Purpose of Document

This document describes the software interface to the AES "Naples" PC Lamp/ Reel/ Switch Controller (PLRSC) User Manual as seen by a software engineer writing in either the C or C++ programming languages on the PC.

Intended Audience

The intended audience of this document is the software engineers who will be writing software on the PC that will use the PLRSC card to drive reels and lamps and read switches.

Document Layout

The document itself is split into a number of sections. Within each section, there are three sections.

- **Operational Overview.**
  Where the way in which this area is intended to work is explained.
- **Function Definitions.**
  Where you will find exact details on each function call.
- **Usage Details.**
  This gives details on exactly how the PLRSC system operates.

The first two sections are intended to reflect different levels of complexity at which an initial game programmer may wish to use the interface.

1. Basic Operation

   These are the minimum set of “vanilla” functions that may be used to get a working *demonstration* program running, using a pre-defined set of hardware.

   Using these calls alone; the software engineer can write a working program and get a feel for the ease with which he can control the lamp and switches attached to his game.

2. Full Details

   These build on the set of functions provided within the “Basic Operation” section. They explain how the programmer can describe the hardware their game uses. (The basic operation section relies on the fact that the development kit hardware is described by the default configuration.)

3. Utility Functions

   These miscellaneous functions are concerned with the administration of the game system.

Getting Started

Installation

The PLRSC unit is a standard USB 1.1 peripheral. Installation of the OS driver is, as with any USB peripheral, when the unit is detected the user is prompted to insert the installation CD. This CD will install the low level drivers, which may be removed using the standard “Add or Remove Programs” facility for the “AES PLRSC USB Drivers” entry.

In addition, two other steps need to be undertaken, at present manually:

- The interface AESPLRSC.DLL needs to be copied from the installation CD to Windows\System32
- The High Level driver program needs to be copied from the installation CD to a convenient folder, and an entry made in the Startup folder to run this at system boot.

Operation

A PLRSC is required for two reasons:

1. To provide the electrical interface to the lamps reels and switches.
2. To execute the high frequency real time code associated with driving such peripherals.

The PLRSC unit contains an embedded processor that is responsible for providing the real time control of the peripherals.

The division between the PLRSC embedded controller and the PC application is defined to be at the level of the 18 millisecond basic Windows timer interrupt. Processing therefore splits into:

- Less than 18 milliseconds - embedded processor
    1. Driving the reel stepper motor phases to obtain smooth acceleration, deceleration and running.
    2. Driving the multiplexed array of 16 x 16 drivers to turn each individual lamp on or off for a complete millisecond.
    3. Selecting individual lamps on or off in successive millisecond intervals so as to obtain a intensity level.
    4. Scanning the switch matrix to determined if individual switches are open or closed.

- More than 18 milliseconds - Windows Application
    1. Choosing if a reel should be running, stopping at a position or “nudging” from one position to the next.
    2. Choosing the intensity level of individual lamps to provide player feed back, changing “attract sequences”, etc.
    3. Specifying the exact way in which reel motors should be controlled (Ramp Tables)
    4. Specifying the exact way in which lamp intensity should be achieved. (See later.)

The interface between the embedded programmer and the PC is **not** done as a series of events and commands, as these can result in mysterious artefacts if they are not processed at the speed at which they are issued.

The interface from the embedded processor to the PC is expressed as a set of states that are continually updated to show the PC the **current** state of the hardware.

The interface from the PC to the embedded processor is expressed as a set of states that show the embedded processor what the PC would like the hardware to be doing at the time that they are changed.

As an example, spinning a reel from 5 to 20 is performed as:

- Application state: At position 5 => Run forward.
  Reel state: Stopped at 5.

- Application state: Run Forward
  Reel state: Stopped at 5 => Running forward.

- Application state: Run forward => At position 20
  Reel state: Running forward.

- Application state: At position 20
  Reel state: Running forward => Stopping at 20

- Application state: At position 20
  Reel state: Stopping at 20 => Stopped at 20

The following function calls are provided to implement a minimum system. Using the functions described within this section, one can spin reels, control lamps and access switches.

Open PLRSC

Synopsis
This call is made by the PC application software to open the “"PC Lamp/Reel/Switch Controller” Interface.

```
long OpenPLRSC (void);
```

Parameters
None

Return Value
If the Open call succeeds then the value zero is returned.

In the event of a failure one of the following standard windows error codes will be returned, either as a direct echo of a Windows API call failure, or to indicate internally detected failures that closely correspond to the quoted meanings.

Error Number	**Suggested string for English decoding**	**Microsoft Mnemonic**	**Retry**
13	The DLL, application or device are at incompatible revision levels.	ERROR_INVALID_DATA	No
20	The system cannot find the device specified.	ERROR_BAD_UNIT	No
21	The device is not ready.	ERROR_NOT_READY	Yes
31	Driver program not running.	ERROR_GEN_FAILURE	Yes
170	The USB link is in use.	ERROR_BUSY	Yes
1167	The device is not connected.	ERROR_DEVICE_NOT_CONNECTED	Yes

Remarks

1. With a USB system, there is a noticeable time for the USB communications to start. This may cause error returns labelled “Yes” under Retry in the above table. This indicates that the call to **OpenPLRSC** should be retried periodically until it has failed for at least 5 consecutive seconds before deciding that the interface is actually inoperative.

2. Whereas an Open service normally requires a description of the item to be opened (and returns a reference to that Item) there is only one PLRSC Interface unit in a system. Hence, any “Open” call must refer to that single item.

SetLampIntensity

Synopsis
The SetLampIntensity call is used by the PC application to control the individual lamps in the matrix connected to the interface.

```
void SetLampIntensity (long IndicatorNumber,
                       long Intensity) ;
```

Parameters
IndicatorNumber
    This is the number of the Lamp that is being controlled.
Intensity
    This is a number in the range 0 (off) to 31 (full on) that selects the desired intensity.

Return Value
    None

Remarks
1. Intensities are more fully described below.

MatrixSwitchOpens / MatrixSwitchCloses

Synopsis
The calls to **MatrixSwitchOpens** and **MatrixSwitchCloses** are made by the PC application to read the state of individual switches in the matrix connected to the interface.

```
long MatrixSwitchOpens  (long SwitchNumber) ;
long MatrixSwitchCloses (long SwitchNumber) ;
```

Parameters
SwitchNumber
  This is the number of the switch that is being controlled. In principle the API can support 256 switches, though note that the current unit only supports 128.

Return Value
The number of times that the specified switch has been observed to open or to close, respectively.

Remarks
1. The convention is that at initialisation time all switches are open, a switch that starts off closed will therefore return a value of 1 to a SwitchCloses call immediately following the OpenPLRSC call.

2. The expression (SwitchCloses(n) == SwitchOpens(n)) will always return 0 if the switch is currently closed and 1 if the switch is currently open.

3. Repeat pressing / tapping of a switch by a user will be detected by an increment in the value returned by SwitchCloses or SwtichOpens.

4. The user only needs to monitor changes in one of the two functions (in the same way as most windowing interfaces only need to provide functions for button up or button down events).

SetReelState

Synopsis

The **SetReelState** call is used by the PC application to control a reel connected to the interface.

```
void SetReelState (long ReelNumber,
                   ReelState State,
                   long Symbol) ;
```

Parameters

ReelNumber

This is the number (0 - N-1) of the Reel that is being controlled.

State

This is the state that the application desires the reel to be in, it is chosen from one of the following:

0	INVALID	The initial value - corresponding to no requested state.
1	CALIBRATE	Spin the reel so as to establish the location of symbol one.
2	STOP	Make the moving reel stop at the symbol specified.
3	NUDGE_FORWARD	Move the stationary reel one symbol to the adjacent position in the forward direction.
4	NUDGE_BACKWARD	Move the stationary reel one symbol to the adjacent position in the backward direction.
5	SPIN_FORWARD	Start the reel spinning forward.
6	SPIN_BACKWARD	Start the reel spinning backward.
7	LOCK_REEL	“Twitch” the reel to simulate mechanical locking.
8	SHAKE_REEL	Start the reel shaking.
9	STOP_SHAKE	Stop a shaking reel.

ReelNumber

This is the symbol number (1 - N) required for some of the states.

Return Value

None

Remarks

2. Some of the states do not make sense for some of the states of the reel - in these circumstances an INVALID_REQUEST status will be returned for the GetReelState call.

GetReelSymbol

Synopsis

The **GetReelSymbol** call is used by the PC application to control a reel connected to the interface.

```
long GetReelSymbol (long ReelNumber) ;
```

Parameters

ReelNumber

This is the number (0 - N-1) of the Reel whose position is being requested.

Return Value

If positive, the number of the symbol at which the reel is either stopped or committed to stop at.
If negative, the number of the symbol that the spinning reel is currently passing.

Remarks

GetReelState

Synopsis
The **GetReelState** call is used by the PC application to control a reel connected to the interface.

```
ReelState GetReelState (long ReelNumber) ;
```

Parameters
ReelNumber
This is the number (0 - N-1) of the Reel whose position is being requested.

Return Value
This is the state that the reel is currently actually in, it is chosen from one of the following:

0	INVALID	The reel cannot be made to comply with the currently requested state.
1	CALIBRATE	The reel is spinning so as to establish the location of symbol one.
2	STOPPING	The reel will stop at the position that has been specified.
3	NUDGE_FORWARD	The reel is moving the reel one symbol to an adjacent position in the forward direction.
4	NUDGE_BACKWARD	The reel is moving the reel one symbol to an adjacent position in the backward direction.
5	SPIN_FORWARD	The reel is spinning forward
6	SPIN_BACKWARD	The reel is spinning backward
7	LOCK_REEL	The reel is “locking”.
8	SHAKE_REEL	The reel is shaking.
9	STOP_SHAKE	The reel is stopping shaking.
10	UNTRIED_CALIBRATION	The reel position is unknown because calibration has not yet been tried.
11	FAILED_CALIBRATION	The reel position is unknown because calibration *has* been tried, but has failed.
12	LOST_CALIBRATION	The “sanity checks” on the opto tab have failed and the PLRSC no longer knows where the reel is.
13	PENDING	The last command has not yet been actioned by the embedded controller.
14	STOPPED	The reel is stopped at a symbol (see **GetReelPosition**)

VFD_Brightness

Synopsis
The **VFD_Brightness** call is used by the PC application to set the brightness level of the Vacuum Fluorescent Display connected to the Naples interface.

```
void VFD_Brightness(long Brightness) ;
```

Parameters
Brihghtness
This is the VFD brightness level (0 - 31) being requested.

Return Value
None

VFD_String

Synopsis
The **VFD_String** call is used by the PC application to send an ASCII string to the Vacuum Fluorescent Display connected to the Naples interface.

```
void VFD_String(char * String) ;
```

Parameters
String
This is a pointer to the string to be displayed.

Return Value
None

Remarks

Remarks
1. The VFD is actually a 16-character device, but the interface allows for up to 32 characters in the string.

2. If the string is greater than 16 characters, only the first 16 will be displayed.

Getting Started Code Examples

The following code fragments are intended to provide clear examples of how the calls to the PLRSC are designed to be used. Each function will provide the central processing for a small command line demonstration program.

General Framework Example

```
int Main()
    {
    long OpenStatus = OpenPLRSC() ;
    if (OpenStatus != 0)
        {
        printf("PLRSC open failed - %ld\n", OpenStatus) ;
        }
    else
        {
        // Then the open call was successful
        DoExampleCode() ;
        }
    }
```

Switch Example

```
void Switches(void)
    {
    int Current[128] ;
    int Switch ;
    int Repeat ;

    for (Switch = 0 ; Switch < 128 ; ++Switch)
        {
        Current[Switch] = MatrixSwitchCloses(Switch) ;
        }


    for (Repeat = 0 ; Repeat < 200 ; ++Repeat)
        {
        for (Switch = 0 ; Switch < 128 ; ++Switch)
            {
            if (Current[Switch] != MatrixSwitchCloses(Switch))
                {
                printf("Switch %d pushed\n", Switch) ;
                Current[Switch]++ ;
                }
            }
        Sleep(50) ;
        }
    }
```

Lamp Example

```
void Lamps(void)
    {
    int Lamp ;

    for (Lamp = 0 ; Lamp < 256 ; ++Lamp)
        {
        SetLampIntensity(Lamp, 0) ;
        }

    for (Lamp = 0 ; Lamp < 256 ; ++Lamp)
        {
        if (Lamp > 0)
            {
            SetLampIntensity(Lamp - 1, 0) ;
            }

        SetLampIntensity(255 - (Lamp - 1), 0) ;

        SetLampIntensity(Lamp      , 31) ;
        SetLampIntensity(255 - Lamp, 15) ;
        Sleep(100) ;
        }
    }
```

Reel Example

```
void ReelSpinExample(void)
    {
    int Position ;
    int Reel ;
    int Completed ;
    //
    // First we have to calibrate the reel
    //

    // Start the process
    for (Reel = 0 ; Reel < 4 ; ++Reel)
        {
        SetReelState(Reel, CALIBRATE, 0) ;
        }

    // and wait for it to complete
    do  {
        Completed = 0 ;
        for (Reel = 0 ; Reel < 4 ; ++Reel)
            {
            if (GetReelState(Reel) != CALIBRATE)
                {
                ++Completed ;
                if (GetReelState(Reel) != STOP)
                    {
                    printf("Reel %d failed to calibrate\n", Reel) ;
                    }
                }
            }
        Sleep(100) ;
        } while (Completed < 4) ;

    //
    // Now we spin each reel to each position in turn
    //
    for (Position = 1 ; Position <= 16 ; ++Position)
        {
        // Start them all spinning
        for (Reel = 0 ; Reel < 4 ; ++Reel)
            {
            SetReelState(Reel, SPIN_FORWARD, 0) ;
            }


        // Now, at two second intervals, stop at the new position
        for (Reel = 0 ; Reel < 4 ; ++Reel)
            {
            Sleep(2000) ;
            SetReelState(Reel, STOP, Position) ;
            }


        // Wait for all 4 reels to stop spinning
        do  {
            Completed = 0 ;
            for (Reel = 0 ; Reel < 4 ; ++Reel)
                {
                if (GetReelState(Reel) != SPIN_FORWARD)
                    {
                    ++Completed ;
                    }
                }
            Sleep(100) ;
            } while (Completed < 4) ;

        // And leave it 4 seconds before re-starting
        Sleep(4000) ;
        }
    }
```

Full Game System

Background

When implementing an actual game, both the reels and lamps have a number of characteristics that are variable enough that they require “tuning” by an engineer in order to produce acceptable functionality.

Lamp characteristics.

For a lamp, the characteristics include the thermal inertia, voltage / temperature response and human eye light perception, which combine in unpredictable ways to give the drive patterns that correspond to perceived levels.

The multiplexed array of 16 x 16 lamps is driven at the lowest level so as to turn each individual lamp in the array on or off for a complete millisecond. (In fact, it is turned on for 1/16 of the millisecond, but at 16x the normal full power.)

Variable intensity on each lamp is achieved by selecting the lamp as on or off in successive millisecond intervals. The PLRSC repeats the selected pattern for a lamp every 32 milliseconds. On - full intensity - is (obviously) achieved by driving the lamp for each of the 32 milliseconds, off is achieved by never turning the lamp on.

With the exception of 0, the intensity level is one less than the number of milliseconds for which the lamp is on, thus intensity level 31 is on for 32 out of every 32 milliseconds, intensity level 15 is on every other millisecond - for 16 out of every 32.

Reel characteristics.

The reels driven by this reel controller have stepper motors. These are operated by supplying current to motor coils and changing phase to move the reel from one position to the next.

The reel controller needs to know the number of steps for a complete revolution and the number of steps the motor turns to move from one symbol to the next. (The total number of symbols is then obviously one divided by the other.)

Factors such as the inertia of the reel and the torque of the motor combine to give each reel a different set of phase timing requirements during acceleration, spinning, deceleration and nudging. A definition of these is generally known as a “Ramp Table”.

The values within the ramp table are typically timings (in milliseconds) between the application of successive phase patterns to the motor coils to achieve the particular required operation.

SetReelCharacteristics

Synopsis
The **SetReelCharacteristics** call is used by the PC application to control the individual lamps in the matrix connected to the interface.

```
void SetReelCharacteristics (long Reel,
                             RampTableDef* RampTable) ;
```

Parameters
Reel
This is the index (0-7) of the reel being defined.

RampTable
The ramp table definition to use for this reel. See below for a description of the structure.

Return Value
None

Remarks
1. The PLRSC ships with a default (test) set of definitions. This has to accurately match the characteristics of the reels and so will typically be replaced by such a description *before* the reels are calibrated or operated.

RampTableDef

A “RampTableDef” structure holds the information necessary to properly drive a reel of a particular type.

```
typedef struct
  {
  unsigned char  Speed ;
  unsigned char  Steps ;
  unsigned char  StepsPerSymbol ;
  unsigned char  DecelerationStates ;
  unsigned short Flags ;
  unsigned char  LockFrequency[2] ;
  unsigned char  ShakeFrequency[4] ;
  unsigned char  NudgeDescriptor[8] ;
  unsigned char  AccelerationTable[64] ;
  unsigned char  DecelerationTable[64] ;
  }  RampTableDef ;
```

The fields within this structure have the following meanings.

Fields

Speed

This is the time in milliseconds between successive phase changes when a reel is spinning normally (i.e. neither accelerating or decelerating).

For example, a 48-step reel running at 96 RPM would have a “Speed” field of 13.

Steps

This is the number of steps (phase changes) required for the stepper motor to make one complete revolution.

StepsPerSymbol

This is the number of steps (phase changes) required for the stepper motor to move from one symbol to the next.

DecelerationStates

This is the number of steps (phase changes) in the deceleration sequence. This is used by the reel controller to determine the phase at which a deceleration sequence must begin in order for the reels to stop at the desired position.

Flags

This field holds a number of Boolean flags:

FLAG_SHAKE_REEL_ON_STOP	0x0001
FLAG_SHAKE_REEL_IN_SYNC	0x0002
FLAG_NEGATIVE_HOME_PULSE	0x0004
FLAG_NUDGE_IN_HALF_STEPS	0x0008
FLAG_BRAKING_PHASE	0x0010

- The first flag causes the reel to “twitch” when *it* stops.
- The second flag causes the reel to “twitch” in synchronisation with *other* stopping reels. Together, the two preceding flags are used in an attempt to make the reel appear more “mechanical”.
- The next flag specifies the high/ low sense of the “home” pulse.
- The next flag specifies whether nudging takes place in full or half steps.

- The next flag specifies whether the penultimate phase in a deceleration is a "braking" phase. In this case the last-but-one step during deceleration involves the application of a reverse phase to the motor to help stop it "dead".

LockFrequency

This four-byte field holds two values (in milliseconds) that are the timing values for the phase changes during a "lock" action. A lock action would be used either to:

1. Simulate mechanical locking when a "hold" button is operated.
2. Simulate mechanical coupling when other reels in a reel set come to a halt.

ShakeFrequency

This four-byte field holds four values (in milliseconds) that are the timing values for the phase changes during a "shake" cycle. A shake cycle involves applying four short phase changes to the coils to pull the reel a short distance in either direction.

NudgeDescriptor

This eight byte, zero-padded field holds the values (in milliseconds) that are the timing values for the phase changes during a single nudge operation.

AccelerationTable

This sixty-four byte, zero-padded field holds the values (in milliseconds) that are the timing values for the phase changes during a complete acceleration sequence.

DecelerationTable

This sixty-four byte, zero-padded field holds the values (in milliseconds) that are the timing values for the phase changes during a complete deceleration sequence.

Normally, the number of non-zero entries in this table should match the value in the "DecelerationStates" field. However, in the case where a "reverse braking phase" is included in the deceleration sequence, the "DecelerationStates" field should hold one less than the number of entries in the deceleration table.

Note that the table allows for long deceleration sequences. If the number of deceleration phases is specified as being greater than the number of steps for a complete revolution, correct deceleration operation should not be expected.

Utility Functions

PLRSCFirmwareVersion

Synopsis
The PLRSC**FirmwareVersion** call allows a control application to discover the exact description of the firmware running on the unit.

```
long PLRSCFirmwareVersion (char* CompileDate,
                           char* CompileTime);
```

Parameters
1. CompileDate
   This is a pointer to a 16 byte area that receives a null terminated printable version of the date on which the firmware was installed.
2. CompileTime
   This is a pointer to a 16 byte area that receives a null terminated printable version of the time at which the firmware was installed.

Return Values
The firmware version, as a 32 bit integer. This is normally shown as 4 x 8 bit numbers separated by dots.

Remarks
Either or both of the character pointers may be null.

PLRSCDriverStatus

Synopsis
The PLRSCDriverStatus call allows an interested application to retrieve the status of the USB Driver program for PLRSC system.

```
long PLRSCDriverStatus (void) ;
```

Parameters
None

Return Values

Mnemonic	**Value**	**Meaning**
USB_IDLE	0	No driver or other program running
STANDARD_DRIVER	1	The driver program is running normally
FLASH_LOADER	2	The flash re-programming tool is using the link
MANUFACTURING_TEST	3	The manufacturing test tool is using the link
DRIVER_RESTART	4	The standard driver is in the process of exiting / restarting
USB_ERROR	5	The driver has received an error from the low level driver

Remarks

1. Be aware that further error statuses may be added. Any response other than STANDARD_DRIVER should be regarded as indicating that the system is not currently functional.

PLRSCDriverExit

Synopsis

The **PLRSCDriverExit** call allows a control application to request that the USB driver program exits in a controlled manner.

```
void PLRSCDriverExit (void) ;
```

Parameters

None

Return Values

None

Remarks

This sets the **PLRSCDriverStatus** to DRIVER_RESTART.

Disclaimer

This manual is intended only to assist the reader in the use of this product and therefore Aardvark Embedded Solutions shall not be liable for any loss or damage whatsoever arising form the use of any information or particulars in, or any incorrect use of the product. Aardvark Embedded Solutions reserve the right to change product specifications on any item without prior notice