ViperBoard API Specification

Nano River Technologies
October 2010

Table of Contents

ABBREVIATIONS

API	Application Programming Interface
GPIO	General Purpose IO
GPIOA	ViperBoard GPIO Port A (advanced GPIO interface)
GPIOB	ViperBoard GPIO Port B (digital IO interface)
I2C	Inter-Integrated Circuit
IIC	Inter-Integrated Circuit (same as I2C)
IO	Input / Output
Master	An interface which supplies the clock like the SPI master or I2C master on ViperBoard
NRT	Nano River Technologies
PWM	Pulse Width Modulation
Slave	An interface which receives the clock like the SPI slave or I2C slave on ViperBoard
SPI	Serial Peripheral Interface
USB	Universal Serial Bus

1. Introduction

This document provides the software interface (API) available for the ViperBoard.

ViperBoard is one of the most feature rich interfacing boards available for your PC including:

- high speed SPI master and slave (up to 17 channels)
- high speed I2C master and slave with optional pull-ups
- 16 bit GPIO Port A capable of digital I/O, interrupts, PWM
- optional low pass filtering for 2 GPIO Port A pins for analogue output
- 16 bit GPIO Port B capable of digital I/O
- 4 bit analogue input

Chapter 3 provides a high level summary of all software tasks, grouped by interface. Chapter 4 is a summary of what functions would normally get called and in what order for some common user scenarios. Chapters 5 to 14 provide in-depth description of every function available to ViperBoard.

In chapter 15 pre-defined types and constants declared in the API are summarised.

2. File Summary

To follow are the main files which together make up the ViperBoard API. In order to see how these should be used in a real application, please refer to the "**ViperBoard Application Examples"** document.

viperboard.h:	This is an include file for the ViperBoard. It contains types and constants that you can refer to in calling the tasks/functions.
viperboard_class.h:	This is the header file for the ViperBoard Class Library.
viperboard_class.cpp:	This is the C++ file for the ViperBoard Class Library. The file calls functions in the OpenSource libusb_0.dll.
libusb.h:	This is the header file to the OpenSource libusb_0.dll.

3. Task Overview

High level tasks are provided for each of the physical interfaces – including general purpose I/O (GPIO), I2C, SPI and analogue inputs. This section provides a high level summary list of tasks available per interface.

3.1. USB Tasks

- **OpenDevice** This function is simply used to open a connection to the ViperBoard.
- **Nano_Revision** This function returns the version of the ViperBoard firmware.

3.2. Events

- **Nano_GetEvents** This function reports GPIO, SPI slave or I2C slave events.

3.3. GPIO Port A Tasks

- **Nano_GPIOASetContinuousMode** This function can be used to set a GPIO as continuous pulsed (port A).
- **Nano_ GPIOASetPulseMode** This function can be used to set a GPIO as a one-shot pulse (port A).
- **Nano_ GPIOASetPWMMode** This function can be used to set a GPIO as PWM of some level (port A).
- **Nano_ GPIOASetDigitalOutputMode** This function can be used to set a GPIO as a digital output (port A).
- **Nano_ GPIOASetDigitalInputMode** This function can be used to set a GPIO as a digital input (port A).
- **Nano_ GPIOASetInterruptInputMode** This function can be used to set a GPIO as an interrupt input (port A).
- **Nano_ GPIOAGetDigitalInput** This function returns the value of a GPIO pin (port A).

3.4. GPIO Port B Tasks

- **Nano_GPIOBSetDirection** This function can be used to set the I/O direction for all GPIOs (port B).
- **Nano_GPIOBGetDirection** This function returns the I/O direction for all GPIOs (port B).
- **Nano_GPIOBWrite** This function sets the output level for all GPIOs (port B).
- **Nano_GPIOBRead** This function returns the logic level for all GPIOs (port B).
- **Nano_GPIOBSetSingleBitDirection** This function sets direction for one specified GPIO (port B).
- **Nano_GPIOBGetSingleBitDirection** This function returns direction for one specified GPIO (port B).
- **Nano_GPIOBSingleBitWrite** This function sets the output level for one specified GPIO (port B).
- **Nano_GPIOBSingleBitRead** This function returns the logic level for one specified GPIO (port B).

3.5. I2C Master Tasks

- **Nano_I2CMasterSetFrequency** This function is used to set the line rate for I2C master transfers.
- **Nano_I2CMasterScanConnectedDevices** This function returns all slave device IDs connected to the I2C bus.
- **Nano_I2CMasterWrite** This function performs a single I2C master write transaction.
- **Nano_I2CMasterRead** This function performs a single I2C master read transaction.
- **Nano_I2CMasterWriteRead** This function performs a back-to-back I2C write then read.
- **Nano_I2CMasterReadWrite** This function performs a back-to-back I2C read then write.
- **Nano_I2CMasterWriteWrite** This function performs two back-to-back I2C writes.
- **Nano_I2CMasterReadRead** This function performs two back-to-back I2C read.

3.6. I2C Slave Tasks

- **Nano_I2CSlaveConfig** This function can be used to set the slave device ID .
- **Nano_I2CSlaveArm** This function can be used to arm the ViperBoard I2C slave.
- **Nano_I2CSlaveBuffer1Write** This function writes to I2C slave buffer 1.
- **Nano_I2CSlaveBuffer2Write** This function writes to I2C slave buffer 2.
- **Nano_I2CSlaveBuffer1Read** This function reads from I2C slave buffer 1.
- **Nano_I2CSlaveBuffer2Read** This function reads from I2C slave buffer 2.

3.7. SPI Master & Slave Tasks

- **Nano_SPIConfigure** This function is used to configure SPI channels as master or slave.
It also allows setting of the chip select sense, CPOL and CPHA settings.

3.8. SPI Master Tasks

- **Nano_SPIMasterSetFrequency** This function can be used to set the line rate for SPI master transfers.
- **Nano_SPIMasterReadWrite** This function can be used to perform a master SPI read/write.
- **Nano_SPIMasterWrite** This function can be used to perform a master SPI write only.
- **Nano_SPIMasterRead** This function can be used to perform a master SPI read only.
- **Nano_SPIMasterReadWrite4** Speed optimised version of Nano_SPIMasterReadWrite().

3.9. SPI Slave Tasks

- **Nano_SPISlaveArm** This function can be used to arm the ViperBoard SPI slave.
- **Nano_SPISlaveBufferWrite** This function writes to one of the SPI slave buffers.
- **Nano_ SPISlaveBufferRead** This function reads from one of the SPI slave buffers.

3.10. Analogue Input Tasks

- **Nano_ADCRead** This function can be used to read an analogue input.

4. Common User Scenarios

There are very many API functions available to ViperBoard applications. This section attempts to show some of the commonly used sequences. Of course one should consult the later sections of this document to find out exactly how the functions work. The scenarios are aimed at providing a starting point for users to develop their own applications. More examples can also be found in the "**ViperBoard Application Examples**" document.

GPIO Applications

OpenDevice()	Open connection to ViperBoard.
↓	
Nano_GPIOASetContinuousMode() Nano_GPIOASetPulseMode() Nano_GPIOASetPWMMode() Nano_GPIOASetDigitalOutputMode() Nano_GPIOASetDigitalInputMode() Nano_GPIOASetInterruptInputMode()	Configure the GPIO mode (Port A advanced GPIOs).
Nano_GPIOBSetDirection() Nano_GPIOBSetSingleBitDirection() Nano_GPIOBWrite() Nano_GPIOBSingleBitWrite()	Configure the GPIO mode (Port B GPIOs).
↓	
Nano_GPIOAGetDigitalInput() Nano_GPIOBRead() Nano_GPIOBSingleBitRead()	Read GPIO inputs during the application.
↓	
Nano_GetEvents()	Poll for GPIO events.

SPI Master Applications

OpenDevice()	Open connection to ViperBoard.
↓	
Nano_SPIConfigure()	Configure the SPI channels with respect to master/slave, chip select sense, CPOL and CPHA.
Nano_SPIMasterSetFrequency()	Set the line rate.
↓	
Nano_SPIMasterReadWrite() Nano_SPIMasterWrite() Nano_SPIMasterRead()	Perform an SPI transaction.

SPI Slave Applications

OpenDevice()	Open connection to ViperBoard.
↓	
Nano_SPIConfigure()	Configure the SPI channels with respect to master/slave, chip select sense, CPOL and CPHA.
↓	
Nano_SPISlaveBufferWrite()	Write to the SPI slave buffer.
↓	
Nano_SPISlaveArm()	Arm the SPI slave.
↓	
Nano_GetEvents()	Poll for SPI slave events.
↓	
Nano_SPISlaveBufferRead()	Read the slave SPI buffer based on the channel and data length.

I2C Master Applications

OpenDevice()	Open connection to ViperBoard.
↓	
Nano_I2CMasterSetFrequency()	
↓	
Nano_I2CMasterScanConnectedDevices()	List all the I2C devices connected on the bus.
↓	
Nano_I2CMasterWrite() Nano_I2CMasterRead() Nano_I2CMasterWriteRead() Nano_I2CMasterReadWrite() Nano_I2CMasterWriteWrite() Nano_I2CMasterReadRead()	Perform an I2C master transaction.

I2C Slave Applications

OpenDevice()	Open connection to ViperBoard.
↓	
Nano_I2CSlaveConfig()	Configure the I2C slave device ID.
↓	
Nano_I2CSlaveBuffer1Write()	Fill the slave buffer with response data for an I2C read command.
Nano_I2CSlaveBuffer2Write()	Fill the slave buffer with response data for a second consecutive I2C read command.
↓	
Nano_I2CSlaveArm()	Arm the I2C slave.
↓	
Nano_GetEvents()	Poll for I2C slave events.
↓	
Nano_I2CSlaveBuffer1Read()	Read the data written by an I2C write command.
Nano_I2CSlaveBuffer2Read()	Read the data written by a second consecutive I2C write command.

5. USB Tasks

5.1. OpenDevice

bool **OpenDevice ( )**

This function tries to open a connection to the ViperBoard. If the connection is successful then it returns TRUE, otherwise it will return FALSE. The task should be called at the start of applications before any other ViperBoard functions are called.

5.2. Nano_Revision

WORD **Nano_Revision ( )**

This function returns the revision for the ViperBoard firmware.

6. Events

ViperBoard has the ability to respond to other systems. There are three distinct ways in which this can occur:

GPIO Event:	ViperBoard can receive an interrupt input on one or more of the GPIO Port A pins.
SPI Slave Event:	One or more of the SPI channels can be configured in SPI slave mode and a slave has taken part in an SPI transaction.
I2C Slave Event:	The I2C slave has responded to a command from an external I2C master.

Events should be polled periodically to see if they have occurred. The function *Nano_GetEvents()* allows the user to poll for events and to get more information about what event has occurred.

6.1. Nano_GetEvents

NANO_RESULT **Nano_GetEvents (**
HANDLE *hDevice,*
WORD **pGPIOEvent,*
BOOL **pSPISlaveEvent,*
BOOL **pI2CSlaveEvent,*
BYTE **pSPISlaveChan,*
WORD **pSPISlaveBytes,*
BYTE **pI2CSlaveTransferType,*
WORD **pI2CSlaveTransfer1Bytes,*
WORD **pI2CSlaveTransfer2Bytes*
)

This function is a used to POLL the status of the ViperBoard events. The function handles all event sources including GPIO interrupts, SPI slave transactions and I2C slave transactions.

hDevice: This is the handle to the USB device.

**pGPIOEvent*: This is a 16 bit word which is filled with the interrupt flag for each GPIO bit. Bit 0 corresponds to an interrupt on GPIO_A_00, Bit 15 corresponds to an interrupt on GPIO_A_15. A set bit means an interrupt event occurred. A clear bit means there was no interrupt.

The GPIO event is cleared once *Nano_GetEvents()* is called.

**pSPISlaveEvent*: This bit is used to indicate that the SPI slave was armed and participated in an SPI transfer. TRUE indicates that an event occurred and FALSE indicates an event did not take place. Use **pSPISlaveChannel* and **pSPISlaveBytes* to determine the channel and number of bytes involved in the transfer.

The SPI slave event is ONLY cleared by arming or disarming the SPI slave – i.e. by calling *Nano_SPISlaveArm()*.

**pI2CSlaveEvent:* This bit is used to indicate that the I2C slave was armed and participated in an I2C transfer. TRUE indicates that an event occurred and FALSE indicates an event did not take place. Use **pI2CSlaveTransferType*, **pI2CTransfer1Bytes* and **pI2CTransfer2Bytes* to determine the transfer type and number of bytes involved in the transfer.

The I2C slave event is ONLY cleared by arming or disarming the I2C slave – i.e. by calling *Nano_I2CSlaveArm()*.

**pSPISlaveChan*: Upon receiving an SPI slave event this contains the particular SPI channel involved.

Channel
NANO_SPI_CHANNEL_0
NANO_SPI_CHANNEL_1
. . .
NANO_SPI_CHANNEL_16

**pSPISlaveBytes*: Upon receiving an SPI slave event this contains the number of bytes involved in the slave transfer. For channel 0 this can be up to 4096 bytes. For channels 1-16 this can be up to 256 bytes.

**pI2CSlaveTransferType*: Upon receiving an I2C slave event this contains the particular I2C transfer that took place.

I2C Transfer Type
NANO_IC_WRITE_TRANSFER
NANO_IC_READ_TRANSFER
NANO_IC_WRITEREAD_TRANSFER
NANO_IC_READWRITE_TRANSFER
NANO_IC_WRITEWRITE_TRANSFER
NANO_IC_READREAD_TRANSFER

**pI2CSlaveTransfer1Bytes*: Upon receiving an I2C slave event this contains the number of bytes transferred with command number 1.

**pI2CSlaveTransfer2Bytes*: Upon receiving an I2C slave event this contains the number of bytes transferred with command number 2.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7. GPIO Port A Tasks

In this section the tasks for the Port A of the GPIO interface are described in full detail.

7.1. Nano_GPIOASetContinuousMode

NANO_RESULT **Nano_GPIOASetContinuousMode (**
HANDLE *hDevice,*
BYTE *GPIONumber,*
BYTE *Clock,*
WORD *T1,*
WORD *T2*
)

This function sets one of the GPIO port A pins into continuous pulse mode.

hDevice: This is the handle to the USB device.

GPIONumber: This specifies the GPIO bit that is to be set in continuous mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.

T1: This is used to set the low time of the continuous pulsed output. The duration is in multiples of the clock period. Range is from 1-255.

T2: This is used to set the high time of the continuous pulsed output. The duration is in multiples of the clock period. Range is from 1-255.

Clock: This represents the clock used as the timing unit for the high and low time.

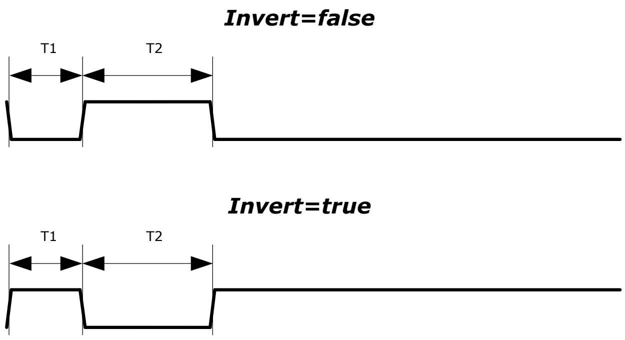
Setting	Base Tick Period
NANO_GPIO_CLK_1	1us
NANO_GPIO_CLK_10	10us
NANO_GPIO_CLK_100	100us
NANO_GPIO_CLK_1000	1ms
NANO_GPIO_CLK_10000	10ms
NANO_GPIO_CLK_100000	100ms

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7.2. Nano_GPIOASetPulseMode

NANO_RESULT **Nano_GPIOASetPulseMode (**
HANDLE *hDevice,*
BYTE GPIONumber,
BYTE Clock,
WORD *T1,*
WORD *T2,*
BOOL *Invert*
)

This function creates a single shot pulse on one of the GPIO A pins.



hDevice: This is the handle to the USB device.

GPIONumber: This specifies the GPIO bit that is to be set in pulse mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.

T1: This is used to set the duration before the pulse starts. The duration is in multiples of the clock period. Range is from 1-255.

T2: This is used to set the duration of the pulse. The duration is in multiples of the clock period. Range is from 1-255.

Invert: This is used to select between active low and active high single shot pulses. (=0) means active high pulse. (=1) means active low pulse.

Clock: This represents the clock used as the timing unit for the high and low time.

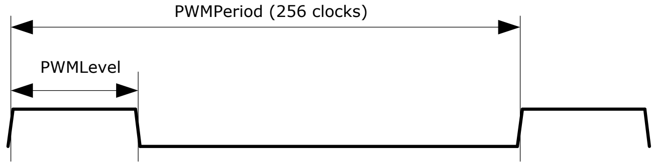
Setting	Base Tick Period
NANO_GPIO_CLK_1	1us
NANO_GPIO_CLK_10	10us
NANO_GPIO_CLK_100	100us
NANO_GPIO_CLK_1000	1ms
NANO_GPIO_CLK_10000	10ms
NANO_GPIO_CLK_100000	100ms

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7.3. Nano_GPIOASetPWMMode

```
NANO_RESULT    Nano_GPIOASetPWMMode (
               HANDLE    hDevice,
               BYTE      GPIONumber,
               BYTE      Clock,
               WORD      PWMLevel
               )
```

This function generates pulse width modulation on a selected GPIO A pin.



hDevice: This is the handle to the USB device.

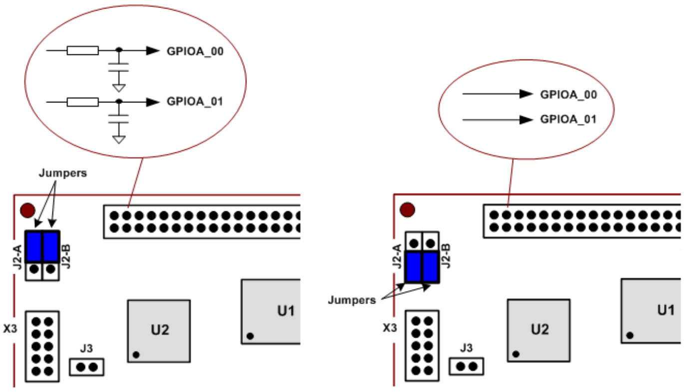
GPIONumber: This specifies the GPIO bit that is to be set in PWM mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.

PWMLevel: This is specifies the high time for the PWM output. The duration is in multiples of the clock period. Range is from 0-255.

Clock: This represents the clock used as the timing unit for the high and low time. It also relates to the PWM period.

Setting	Base Tick Period	PWM Period
NANO_GPIO_CLK_1	1us	256us
NANO_GPIO_CLK_10	10us	2.56ms
NANO_GPIO_CLK_100	100us	25.6ms
NANO_GPIO_CLK_1000	1ms	256ms
NANO_GPIO_CLK_10000	10ms	2.56 sec
NANO_GPIO_CLK_100000	100ms	25.6 sec

For GPIO_A_00 and GPIO_A_01 it is possible to low pass filter the PWM output using a simple on-board RC filter. At the fastest PWM rate the output then becomes an analogue output. The filter is connected using on-board jumper settings. Consult the "ViperBoard Users Guide" to find out exactly how to connect the low pass filter.



The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7.4. Nano_GPIOASetDigitalOutputMode

```
NANO_RESULT     Nano_GPIOASetDigitalOutputMode (
                HANDLE      hDevice,
                BYTE        GPIONumber,
                BOOL        bOutput
                )
```

This function drives one of the GPIO port A pins as a digital output.

hDevice: This is the handle to the USB device.

GPIONumber: This specifies the GPIO bit that is to be set in digital output mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.

bOutput: This specifies the level to drive the GPIO pin. FALSE means drive low. TRUE means drive high.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7.5. Nano_GPIOASetDigitalInputMode

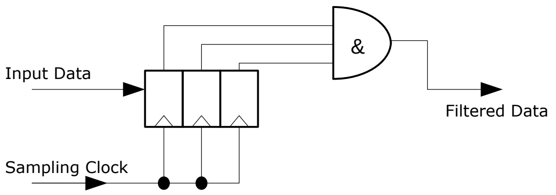
NANO_RESULT **Nano_GPIOASetDigitalInputMode (**
HANDLE *hDevice,*
BYTE *GPIONumber,*
BYTE *Clock*
)

This function makes one of the GPIO port A pins a digital input. Use *Nano_GPIOAGetDigitalInput()* to return the actual level.

hDevice: This is the handle to the USB device.

GPIONumber: This specifies the GPIO bit that is to be set in digital input mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.

Clock: The input circuit includes a simple glitch filter constructed using a shift register and logical AND of the delayed values. The sampling clock for the filter is able to be set using this variable.



Setting	Sampling Clock
NANO_GPIO_CLK_1	1us
NANO_GPIO_CLK_10	10us
NANO_GPIO_CLK_100	100us
NANO_GPIO_CLK_1000	1ms
NANO_GPIO_CLK_10000	10ms
NANO_GPIO_CLK_100000	100ms

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7.6. Nano_GPIOAGetDigitalInput

```
NANO_RESULT    Nano_GPIOAGetDigitalInput (
               HANDLE    hDevice,
               BYTE      GPIONumber,
               BOOL      *pValue
               )
```

This function returns the value of a Port A GPIO pin set as a digital input. The function also works if the pin is configured as a digital output.

hDevice:	This is the handle to the USB device.
GPIONumber:	This specifies the GPIO bit that is to be set in digital input mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.
**pValue*:	This is a pointer to be filled with the input value. TRUE means the input is high and FALSE means the input is low.

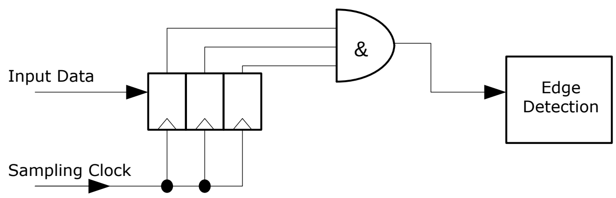
The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

7.7. Nano_GPIOASetInterruptInputMode

```
NANO_RESULT    Nano_GPIOASetInterruptInputMode (
               HANDLE    hDevice,
               BYTE      GPIONumber,
               BOOL      bRiseFall,
               BYTE      Clock,
               )
```

This function makes one of the GPIO port A pins an interrupt input. An interrupt input can be rising-edge or falling-edge triggered. Use *Nano_GetEvents()* to return if an interrupt has occurred and to clear the pending event.

hDevice:	This is the handle to the USB device.
GPIONumber:	This specifies the GPIO bit that is to be set in digital interrupt mode. A value of 0 corresponds to GPIO_A_00. A value of 15 corresponds to GPIO_A_15.
bRiseFall:	This specifies if the interrupt is to be sensitive to a rising-edge or falling-edge interrupt. TRUE is rising-edge and FALSE is falling-edge.
Clock:	The interrupt input circuit includes a simple glitch filter constructed using a shift register and logical AND of the delayed values. This prevents glitches from unnecessarily triggering an interrupt event. The sampling clock for the filter is able to be set using this variable.



Setting	Sampling Clock
NANO_GPIO_CLK_1	1us
NANO_GPIO_CLK_10	10us
NANO_GPIO_CLK_100	100us
NANO_GPIO_CLK_1000	1ms
NANO_GPIO_CLK_10000	10ms
NANO_GPIO_CLK_100000	100ms

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8. GPIO Port B Tasks

In this section the tasks for the Port B of the GPIO interface are described in full detail.

8.1. Nano_GPIOBSetDirection

```
NANO_RESULT    Nano_GPIOBSetBDirection (
               HANDLE    hDevice,
               WORD      Value,
               WORD      Mask
               )
```

This function sets the direction for all 16 bits of GPIO port B.

hDevice:	This is the handle to the USB device.
Value:	This is the required direction. One bit is for each IO. Set (=1) means output, clear (=0) means input. Bit 0 corresponds to GPIO_B_00. Bit 15 corresponds to GPIO_B_15.
Mask:	This is a mask signifying which GPIO direction pins are affected. For a bit to be updated then the corresponding mask bit must be set (=1). Bit 0 corresponds to GPIO_B_00. Bit 31 corresponds to GPIO_B_31.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.2. Nano_GPIOBGetDirection

```
NANO_RESULT    Nano_GPIOBGetDirection (
               HANDLE    hDevice,
               WORD      *pValue,
               )
```

This function returns the direction value for all 16 bits of GPIO port B.

hDevice:	This is the handle to the USB device.
**pValue*:	This is a pointer to be filled with the direction value. One bit is for each IO. A set value bit (=1) means an output and clear (=0) means an input.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.3. Nano_GPIOBWrite

```
NANO_RESULT     Nano_GPIOBWrite (
                HANDLE    hDevice,
                WORD      Value,
                WORD      Mask
                )
```

This function writes to all 16 bits of GPIO port B.

hDevice: This is the handle to the USB device.

Value: This is the required value. ***The GPIO will only be written to if the direction bit is configured for output.*** One bit for each IO. Set (=1) means the GPIO is to be set. Clear (=0) means the GPIO is to be cleared. Bit 0 corresponds to GPIO_B_00. Bit 15 corresponds to GPIO_B_15.

Mask: This is a mask signifying which GPIO pins are affected. For a bit to be updated then the corresponding mask bit must be set (=1). Bit 0 corresponds to GPIO_B_00. Bit 15 corresponds to GPIO_B_15.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.4. Nano_GPIOBRead

```
NANO_RESULT     Nano_GPIOBRead (
                HANDLE    hDevice,
                WORD      *pValue
                )
```

This function returns the value of all 16 bits of GPIO.

hDevice: This is the handle to the USB device.

**pValue*: This is a pointer to be filled with the GPIO values. The GPIO is read irrespective of whether the GPIO bit is set for input or output. Set (=1) means the GPIO is high. Clear (=0) means the GPIO is low. Bit 0 corresponds to GPIO_00. Bit 15 corresponds to GPIO_15.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.5. Nano_GPIOBSetSingleBitDirection

NANO_RESULT **Nano_GPIOBSetSingleBitDirection (**
HANDLE hDevice,
BYTE GPIONumber,
BOOL bOutput
)

This function sets the direction for one of the GPIO port B bits.

hDevice:	This is the handle to the USB device.
GPIONumber:	This specifies which GPIO direction bit to update, 0 means GPIO_B_00 and 15 corresponds to GPIO_15.
bOutput:	This is the required value for the direction bit. TRUE means that the GPIO direction bit should be set for output. FALSE means that the direction bit should be input.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.6. Nano_GPIOBGetSingleBitDirection

NANO_RESULT **Nano_GPIOBGetSingleBitDirection (**
HANDLE hDevice,
BYTE GPIONumber,
BOOL *pbOutput
)

This function returns the direction of one of the GPIO port B bits.

hDevice:	This is the handle to the USB device.
GPIONumber:	This specifies which GPIO direction bit to return, 0 means GPIO_B_00 and 15 corresponds to GPIO_B_15.
**pbOutput*:	This is a pointer to be filled with the direction value. TRUE means that the GPIO direction bit is set for output. FALSE means that the direction bit is set as input.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.7. Nano_GPIOBSingleBitWrite

```
NANO_RESULT    Nano_GPIOBSingleBitWrite (
               HANDLE    hDevice,
               BYTE      GPIONumber,
               BOOL      Value
               )
```

This function sets the value for one of the GPIO port B bits.

hDevice:	This is the handle to the USB device.
GPIONumber:	This specifies which GPIO to update, 0 means GPIO_B_00 and 15 corresponds to GPIO_B_15.
Value:	This is the required value. ***The GPIO will only be written to if the direction bit is configured for output.*** TRUE means that the GPIO direction bit should be driven high. FALSE means that the direction bit should be driven low.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

8.8. Nano_GPIOBSingleBitRead

```
NANO_RESULT    Nano_GPIOBSingleBitRead (
               HANDLE    hDevice,
               BYTE      GPIONumber,
               BOOL      *pValue
               )
```

This function returns the value of one of the GPIO port B bits.

hDevice:	This is the handle to the USB device.
GPIONumber:	This specifies which GPIO value to return, 0 means GPIO_B_00 and 15 corresponds to GPIO_B_15.
**pValue*:	This is a pointer to be filled with the value. TRUE means that the GPIO is high. FALSE means that the GPIO is low.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

9. I2C Master Tasks

In this section the tasks for the I2C Master interface are described in full detail.

9.1. Nano_I2CMasterSetFrequency

NANO_RESULT **Nano_I2CMasterSetFrequency (**
HANDLE *hDevice,*
BYTE *Frequency*
)

This function sets the clock frequency for I2C transfers.

hDevice: This is the handle to the USB device.

Frequency: This is the desired clock frequency for I2C transfers. The following table relates the setting constant to the actual observed clock frequency.

Setting	Clock Frequency
NANO_I2C_FREQ_6MHZ	6MBit/s
NANO_I2C_FREQ_3MHZ	3MBit/s
NANO_I2C_FREQ_1MHZ	1MBit/s
NANO_I2C_FREQ_FAST	400kBit/s (I2C Fast Mode)
NANO_I2C_FREQ_200KHZ	200kBit/s
NANO_I2C_FREQ_STD	100kBit/s (I2C Standard Mode)
NANO_I2C_FREQ_10KHZ	10kBit/s

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

9.2. Nano_I2CMasterScanConnectedDevices

NANO_RESULT **Nano_I2CMasterScanConnectedDevices (**
HANDLE *hDevice,*
DEV_LIST **pList*
)

This function scans all devices on the I2C bus and returns a list of their I2C addresses.

hDevice: This is the handle to the USB device.

**pList*: This is a status list of all I2C devices responding on the I2C bus.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

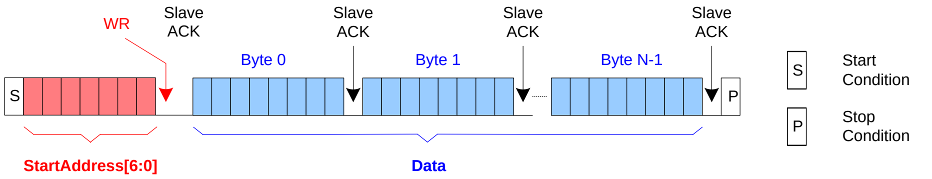
9.3. Nano_I2CMasterWrite

NANO_RESULT **Nano_I2CMasterWrite (**
HANDLE *hDevice,*
BYTE *SlaveAddress,*
WORD *BufferLength,*
BYTE **pOutBuffer*
)

This function initiates a single I2C write transaction by the I2C master. The write transaction can transfer up to 2048 bytes of data.

hDevice:	This is the handle to the USB device.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength:	Sets the buffer length for the transfer. Maximum is 2048 bytes.
**pOutBuffer*:	This is pointer to a buffer which should contain the data to be written.

The function returns NANO_RESULT. The function will return NANO_SUCCESS for a successful write. If the function returns NANO_I2C_PROTOCOL_ERROR then there has been an error in the I2C protocol, for example maybe the slave device has not acknowledged properly. If the function returns a NANO_XACTION_FAILURE, then there is a USB communication failure.



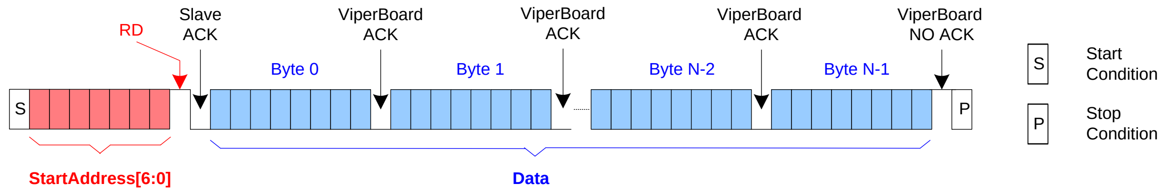
9.4. Nano_I2CMasterRead

NANO_RESULT **Nano_I2CMasterRead (**
HANDLE *hDevice,*
BYTE *SlaveAddress,*
WORD *BufferLength,*
BYTE **pInBuffer*
)

This function initiates a single I2C read transaction by the I2C master. The read transaction can transfer up to 2048 bytes of data.

hDevice:	This is the handle to the USB device.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength:	Sets the buffer length for the transfer. Maximum is 2048 bytes.
**pInBuffer*:	This is pointer to a buffer which is filled with read data.

The function returns NANO_RESULT. The function will return NANO_SUCCESS for a successful read. If the function returns NANO_I2C_PROTOCOL_ERROR then there has been an error in the I2C protocol, for example maybe the slave device has not acknowledged properly. If the function returns a NANO_XACTION_FAILURE, then there is a USB communication failure.



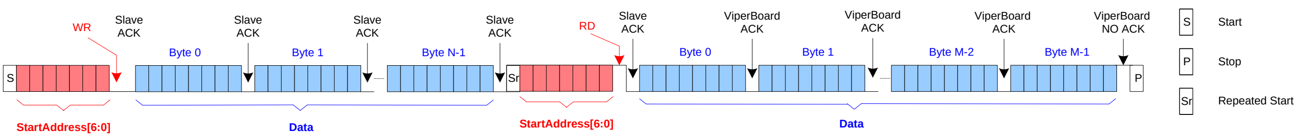
9.5. Nano_I2CMasterWriteRead

NANO_RESULT **Nano_I2CMasterWriteRead (**
HANDLE *hDevice,*
BYTE *SlaveAddress,*
WORD *BufferLength1,*
BYTE **pOutBuffer1,*
WORD *BufferLength2,*
BYTE **pInBuffer2*
)

This function is provided for the master to perform an I2C write (command 1) followed immediately by an I2C read (command 2). The two commands can each transfer 2048 bytes of data and are separated by a repeated start. This particular function can be very useful in talking to I2C EEPROM devices which require setting up the address before doing a read.

hDevice:	This is the handle to the USB device.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the first command. Maximum is 2048 bytes.
**pOutBuffer1*:	This is pointer to a buffer which should contain the data to be written in the first command.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the second command. Maximum is 2048 bytes.
**pInBuffer2*:	This is pointer to a buffer which should be filled with data from the read during the second command.

The function returns NANO_RESULT. The function will return NANO_SUCCESS for a successful write. If the function returns NANO_I2C_PROTOCOL_ERROR then there has been an error in the I2C protocol, for example maybe the slave device has not acknowledged properly. If the function returns a NANO_XACTION_FAILURE, then there is a USB communication failure.



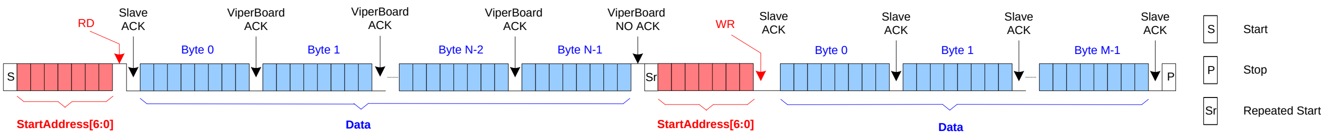
9.6. Nano_I2CMasterReadWrite

NANO_RESULT **Nano_I2CMasterReadWrite (**
HANDLE *hDevice,*
BYTE *SlaveAddress,*
WORD *BufferLength1,*
BYTE **pInBuffer1,*
WORD *BufferLength2,*
BYTE **pOutBuffer2*
)

This function is provided for the master to perform an I2C read (command 1) followed immediately by an I2C write (command 2). The two commands can each transfer 2048 bytes of data and are separated by a repeated start.

hDevice:	This is the handle to the USB device.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the first command. Maximum is 2048 bytes.
**pInBuffer1*:	This is pointer to a buffer which should be filled with data from the read during the first command.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the second command. Maximum is 2048 bytes.
**pOutBuffer2*:	This is pointer to a buffer which should contain the data to be written in the second command.

The function returns NANO_RESULT. The function will return NANO_SUCCESS for a successful write. If the function returns NANO_I2C_PROTOCOL_ERROR then there has been an error in the I2C protocol, for example maybe the slave device has not acknowledged properly. If the function returns a NANO_XACTION_FAILURE, then there is a USB communication failure.



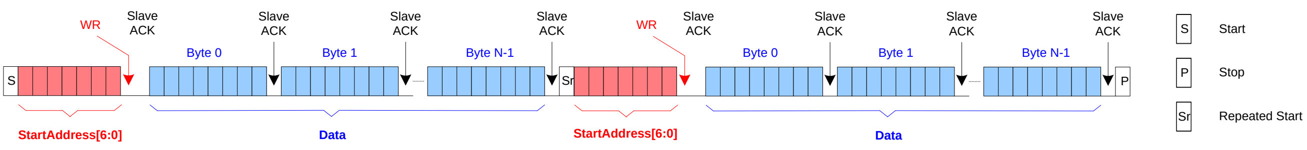
9.7. Nano_I2CMasterWriteWrite

NANO_RESULT **Nano_I2CMasterWriteWrite (**
HANDLE *hDevice,*
BYTE *SlaveAddress,*
WORD *BufferLength1,*
BYTE **pOutBuffer1,*
WORD *BufferLength2,*
BYTE **pOutBuffer2*
)

This function is provided for the master to perform an I2C write (command 1) followed immediately by a second I2C write (command 2). The two commands can each transfer 2048 bytes of data and are separated by a repeated start.

hDevice:	This is the handle to the USB device.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the first command. Maximum is 2048 bytes.
**pOutBuffer1*:	This is pointer to a buffer which should contain the data to be written in the first command.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the second command. Maximum is 2048 bytes.
**pOutBuffer2*:	This is pointer to a buffer which should contain the data to be written in the second command.

The function returns NANO_RESULT. The function will return NANO_SUCCESS for a successful write. If the function returns NANO_I2C_PROTOCOL_ERROR then there has been an error in the I2C protocol, for example maybe the slave device has not acknowledged properly. If the function returns a NANO_XACTION_FAILURE, then there is a USB communication failure.



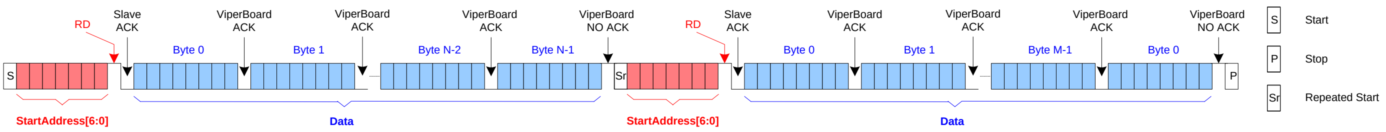
9.8. Nano_I2CMasterReadRead

NANO_RESULT **Nano_I2CMasterReadRead (**
HANDLE *hDevice,*
BYTE *SlaveAddress,*
WORD *BufferLength1,*
BYTE **pInBuffer1,*
WORD *BufferLength2,*
BYTE **pInBuffer2*
)

This function is provided for the master to perform an I2C read (command 1) followed immediately by a second I2C read (command 2). The two commands can each transfer 2048 bytes of data and are separated by a repeated start.

hDevice:	This is the handle to the USB device.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength1:	Sets the buffer length for the first command. Maximum is 2048 bytes.
**pInBuffer1*:	This is pointer to a buffer which should be filled with data from the read during the first command.
SlaveAddress:	This is the I2C slave address (device ID).
BufferLength2:	Sets the buffer length for the second command. Maximum is 2048 bytes.
**pInBuffer2*:	This is pointer to a buffer which should be filled with data from the read during the first command.

The function returns NANO_RESULT. The function will return NANO_SUCCESS for a successful write. If the function returns NANO_I2C_PROTOCOL_ERROR then there has been an error in the I2C protocol, for example maybe the slave device has not acknowledged properly. If the function returns a NANO_XACTION_FAILURE, then there is a USB communication failure.



10. I2C Slave Tasks

In this section the tasks for the I2C Slave interface are described in full detail.

10.1. Nano_I2CSlaveConfig

```
NANO_RESULT    Nano_I2CSlaveConfig (
               HANDLE    hDevice,
               BYTE      SlaveDeviceID
               )
```

This function sets the I2C slave address (device ID) for the ViperBoard I2C slave.

hDevice: This is the handle to the USB device.

SlaveDeviceID: This is the I2C slave address setting for the I2C slave.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

10.2. Nano_I2CSlaveArm

```
NANO_RESULT    Nano_I2CSlaveArm (
               HANDLE    hDevice,
               BOOL      Arm
               )
```

This function is used to arm or dis-arm the ViperBoard I2C slave.

hDevice: This is the handle to the USB device.

Arm: This is the arm bit. When set to TRUE the I2C slave is armed and waiting to receive a command from a connected master. When set to FALSE the slave interface does not respond to any master command.

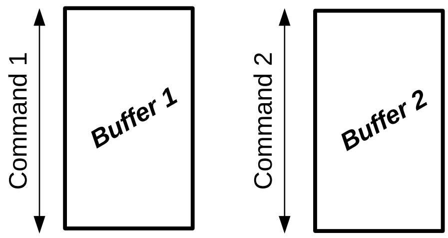
Arming of the I2C slave should take place after the slave ID is configured (Nano_I2CSlaveConfig()) and after any needed response data is filled in the slave data buffers (Nano_I2CSlaveBuffer1Write() and Nano_I2CSlaveBuffer2Write()).

When a slave has responded to a command from the master it will immediately enter a disarmed state. This is to protect any data which is waiting in the slave data buffers. It is up to the application to re-arm the slave when it is ready to respond to a new command.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

10.3. Nano_I2CSlaveBuffer1Write

NANO_RESULT **Nano_I2CSlaveBuffer1Write (**
HANDLE *hDevice,*
WORD *BufferLength1,*
BYTE **pOutBuffer1*
)



The ViperBoard I2C slave has two buffers. The first buffer is provided for storage during single I2C commands such as an I2C Write or and I2C Read. In the case that the slave receives back-to-back commands like I2C Write/Read, I2C Read/Write, I2C Write/Write or I2C Read/Read, then the first command has storage in buffer 1 and the second command has storage in buffer 2.

This command allows the user to fill slave buffer 1. This would be required for example before the slave responds to an I2C master read. Make sure the buffer is filled before the I2C is armed (*Nano_I2CSlaveArm()*).

hDevice:	This is the handle to the USB device.
BufferLength1:	Sets the amount of data to be filled in buffer 1. Maximum is 2048 bytes.
**pOutBuffer1*:	This is pointer to a buffer which should contain the data to be written to buffer 1.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

10.4. Nano_I2CSlaveBuffer2Write

NANO_RESULT **Nano_I2CSlaveBuffer2Write (**
HANDLE *hDevice,*
WORD *BufferLength2,*
BYTE **pOutBuffer2*
)

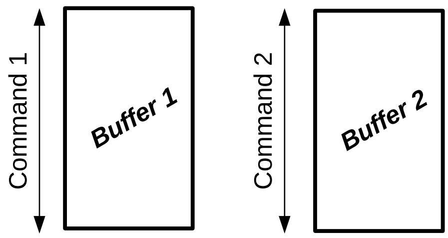
This function is the same as *Nano_I2CSlaveBuffer1Write()* except that it allows filling of buffer 2.

hDevice:	This is the handle to the USB device.
BufferLength2:	Sets the amount of data to be filled in buffer 2. Maximum is 2048 bytes.
**pOutBuffer2*:	This is pointer to a buffer which should contain the data to be written to buffer 2.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

10.5. Nano_I2CSlaveBuffer1Read

NANO_RESULT **Nano_I2CSlaveBuffer1Read (**
HANDLE *hDevice,*
WORD *BufferLength1,*
BYTE **pOutBuffer1*
)



The ViperBoard I2C slave has two buffers. The first buffer is provided for storage during single I2C commands such as an I2C Write or and I2C Read. In the case that the slave receives back-to-back commands like I2C Write/Read, I2C Read/Write, I2C Write/Write or I2C Read/Read, then the first command has storage in buffer 1 and the second command has storage in buffer 2.

This command allows the user to retrieve data from slave buffer 1. This would be required for example after responding to an I2C master write. Make sure the buffer is read before the I2C is re-armed (*Nano_I2CSlaveArm()*).

hDevice: This is the handle to the USB device.

BufferLength1: Sets the amount of data to be read from buffer 1. Maximum is 2048 bytes.

**pInBuffer1*: This is pointer to a buffer which should is filled with data from buffer 1.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

10.6. Nano_I2CSlaveBuffer2Read

NANO_RESULT **Nano_I2CSlaveBuffer2Read (**
HANDLE *hDevice,*
WORD *BufferLength2,*
BYTE **pInBuffer2*
)

This function is the same as *Nano_I2CSlaveBuffer1Read()* except that it allows reading of buffer 2.

hDevice: This is the handle to the USB device.

BufferLength2: Sets the amount of data to be read from buffer 2. Maximum is 2048 bytes.

**pInBuffer2*: This is pointer to a buffer which should is filled with data from buffer 2.

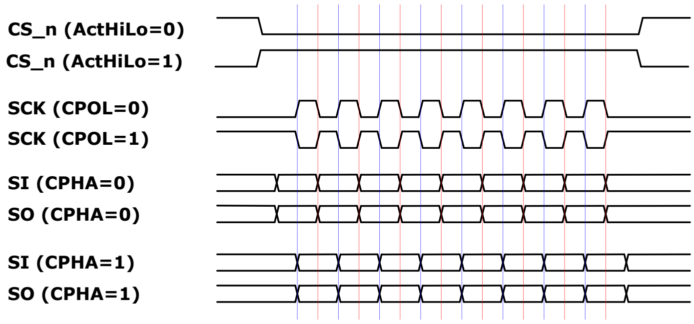
The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

11. SPI Master & Slave Tasks

In this section the tasks which are used for both SPI Master and SPI Slave are described in full detail.

11.1. Nano_SPIConfigure

NANO_RESULT **Nano_SPIConfigure (**
HANDLE *hDevice,*
BYTE *Channel,*
BYTE *Mode,*
BOOL *ActHiLo,*
BYTE *CPOL,*
BYTE *CPHA*
)



The SPI master and slave can facilitate up to 17 separate channels. This is achieved by sacrificing Port A chip GPIO pins as chip selects. This task specifies for each channel if it will be used as an SPI master, SPI slave or left as a GPIO pin. Being left as a GPIO pin is the default from power up. In either SPI master or SPI slave mode then the function allows configuration of the chip select sense (active high/low), SPI clock polarity (CPOL) and SPI clock phase (CPHA).

hDevice: This is the handle to the USB device.

Channel: This is the particular channel number. The following table shows the available channels and the associated chip select when the channel is in either SPI master or SPI slave mode.

Channel	SPI chip select (when in SPI Master or SPI Slave mode)
NANO_SPI_CHANNEL_0	In SPI Master mode the chip select is MST_CS. In SPI Slave mode the chip select is SLV_CS.
NANO_SPI_CHANNEL_1	Chip select used is GPIO_A_00.
NANO_SPI_CHANNEL_2	Chip select used is GPIO_A_01.
NANO_SPI_CHANNEL_3	Chip select used is GPIO_A_02.
. . .	. . .
NANO_SPI_CHANNEL_15	Chip select used is GPIO_A_14.
NANO_SPI_CHANNEL_16	Chip select used is GPIO_A_15.

Mode: This configures the channel as either GPIO, SPI Slave or SPI Master.

Setting	Description
GPIO_MODE	With this selection the channel is not available to the SPI master or slave. The pin is available as a Port A GPIO. This is the DEFAULT.
SPI_SLAVE_MODE	Sets the channel to be used as a SPI Slave. The corresponding GPIO pin becomes the chip select.
SPI_MASTER_MODE	Sets the channel to be used as a SPI Master. The corresponding GPIO pin becomes the chip select.

ActHiLow: In SPI Master or Slave modes, this bit sets the chip select polarity.
- FALSE corresponds to active low chip select.
- TRUE corresponds to active high chip select.

CPOL: In SPI Master or Slave modes, this bit sets the desired polarity for the SPI clock.
- 0 corresponds to low when idle
- 1 corresponds to high when idle

CPHA: In SPI Master or Slave modes, this bit sets the desired phase for the SPI clock.
- 0 corresponds to data valid on leading clock edge
- 1 corresponds to data valid on trailing clock edge

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

12. SPI Master Tasks

In this section the tasks for the SPI Master interface are described in full detail.

12.1. Nano_SPIMasterSetFrequency

NANO_RESULT **Nano_SPIMasterSetFrequency (**
HANDLE *hDevice,*
BYTE *Frequency*
)

This function sets the clock frequency for SPI transfers.

hDevice: This is the handle to the USB device.

Frequency: This is the desired clock frequency for SPI transfers. The following table relates the setting constant to the actual observed clock frequency.

Setting	Clock Frequency
NANO_SPI_FREQ_12MHZ	12MBit/s
NANO_SPI_FREQ_6MHZ	6MBit/s
NANO_SPI_FREQ_3MHZ	3MBit/s
NANO_SPI_FREQ_1MHZ	1MBit/s
NANO_SPI_FREQ_400KHZ	400kBit/s
NANO_SPI_FREQ_200KHZ	200kBit/s
NANO_SPI_FREQ_100KHZ	100kBit/s
NANO_SPI_FREQ_10KHZ	10kBit/s

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

12.2. Nano_SPIMasterReadWrite

NANO_RESULT **Nano_SPIMasterReadWrite (**
HANDLE *hDevice,*
BYTE *Channel,*
BYTE *Length,*
BYTE **pInBuffer,*
BYTE **pOutBuffer*
)

This function provides simultaneous write and read, to and from the SPI slave respectively. The function is valid for a particular chip select channel. In order to use this function, one must specify the particular chip select channel, the length of data to be transferred and fill the output buffer with the data to be sent. Upon competition the input buffer will contain the read data.

hDevice: This is the handle to the USB device.

Channel: This specifies the pin which is to be used as a chip select for the read/write. Be sure to configure that pin as an SPI master using the *Nano_SPIConfigure()* function.

Channel	SPI chip select
NANO_SPI_CHANNEL_0	Chip select used is MST_CS.
NANO_SPI_CHANNEL_1	Chip select used is GPIO_A_00.
NANO_SPI_CHANNEL_2	Chip select used is GPIO_A_01.
NANO_SPI_CHANNEL_3	Chip select used is GPIO_A_02.
. . .	. . .
NANO_SPI_CHANNEL_15	Chip select used is GPIO_A_14.
NANO_SPI_CHANNEL_16	Chip select used is GPIO_A_15.

**pInBuffer*: This is a pointer to a buffer to be filled with data from reading the SPI slave.

**pOutBuffer*: This is a pointer to a buffer to be sent during the write to the SPI slave.

Length: This is the length of the transfer in bytes. All channels can support 4096 bytes maximum.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

12.3. Nano_SPIMasterWrite

NANO_RESULT **Nano_SPIMasterWrite (**
HANDLE *hDevice,*
BYTE *Channel,*
BYTE *Length,*
BYTE **pOutBuffer*
)

This function provides a write only to the SPI slave. The function is valid for a particular chip select channel. In order to use this function, one must specify the particular chip select channel, the length of data to be transferred and fill the output buffer with the data to be sent.

hDevice: This is the handle to the USB device.

Channel: This specifies the pin which is to be used as a chip select for the write. Be sure to configure that pin as an SPI master using the *Nano_SPIConfigure()* function.

Channel	SPI chip select
NANO_SPI_CHANNEL_0	Chip select used is MST_CS.
NANO_SPI_CHANNEL_1	Chip select used is GPIO_A_00.
NANO_SPI_CHANNEL_2	Chip select used is GPIO_A_01.
NANO_SPI_CHANNEL_3	Chip select used is GPIO_A_02.
. . .	. . .
NANO_SPI_CHANNEL_15	Chip select used is GPIO_A_14.
NANO_SPI_CHANNEL_16	Chip select used is GPIO_A_15.

**pOutBuffer*: This is a pointer to a buffer to be sent during the write to the SPI slave.

Length: This is the length of the transfer in bytes. All channels can support 4096 bytes maximum.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

12.4. Nano_SPIMasterRead

NANO_RESULT **Nano_SPIMasterRead (**
HANDLE *hDevice,*
BYTE *Channel,*
BYTE *Length,*
BYTE **pInBuffer*
)

This function provides a read only from the SPI slave. The function is valid for a particular chip select channel. In order to use this function, one must specify the particular chip select channel and the length of data to be transferred. Upon competition the input buffer will contain the read data.

hDevice: This is the handle to the USB device.

Channel: This specifies the pin which is to be used as a chip select for the read/write. Be sure to configure that pin as an SPI master using the *Nano_SPIConfigure()* function.

Channel	SPI chip select
NANO_SPI_CHANNEL_0	Chip select used is MST_CS.
NANO_SPI_CHANNEL_1	Chip select used is GPIO_A_00.
NANO_SPI_CHANNEL_2	Chip select used is GPIO_A_01.
NANO_SPI_CHANNEL_3	Chip select used is GPIO_A_02.
. . .	. . .
NANO_SPI_CHANNEL_15	Chip select used is GPIO_A_14.
NANO_SPI_CHANNEL_16	Chip select used is GPIO_A_15.

**pInBuffer*: This is a pointer to a buffer to be filled with data from reading the SPI slave.

Length: This is the length of the transfer in bytes. All channels can support 4096 bytes maximum.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

12.5. Nano_SPIMasterReadWrite4

NANO_RESULT **Nano_SPIMasterReadWrite4 (**
HANDLE *hDevice,*
BYTE *Channel,*
BYTE *Length,*
BYTE **pInBuffer,*
BYTE **pOutBuffer*
)

This function is exactly the same as Nano_SPIMasterReadWrite() except that it only writes the first four bytes in the InBuffer[]. For some applications it is only necessary to control the first 4 bytes (for example commands to EEPROMs/Flash memories). In this case there can be **significant speed improvement over Nano_SPIMasterReadWrite().**

13. SPI Slave Tasks

In this section the tasks for the SPI Slave interface are described in full detail.

13.1. Nano_SPISlaveArm

```
NANO_RESULT     Nano_SPISlaveArm (
                HANDLE      hDevice,
                BOOL        Arm
                )
```

This function is used to arm or dis-arm the ViperBoard SPI Slave.

hDevice:	This is the handle to the USB device.
Arm:	This is the arm bit. When set to TRUE the SPI Slave is armed and waiting to receive a command from a connected master. When set to FALSE the SPI Slave interface does not respond to any master command.

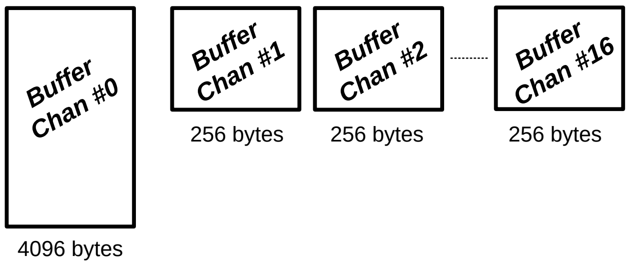
Arming of the SPI slave should take place after any response data is filled in the slave data buffers (Nano_SPISlaveBufferWrite()).

When a slave has responded to a transfer from the master it will immediately enter a disarmed state. This is to protect any data which is waiting in the slave data buffers. It is up to the application to re-arm the slave when it is ready to respond to a new command.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

13.2. Nano_SPISlaveBufferWrite

NANO_RESULT **Nano_SPISlaveBufferWrite (**
HANDLE *hDevice,*
BYTE *Channel,*
WORD *Length,*
BYTE **pOutBuffer*
)



The ViperBoard SPI Slave has a buffer for each channel. Prior to the SPI transfer the buffer contains the data to be sent to the master. After the transaction then the buffer will contain the data written by the master. The buffer for channel 0 contains 4096 bytes. The buffers for channels 1-16 each contain 256 bytes.

This command allows one of the SPI Slave buffers to be written to. This would be required for example before the SPI transaction so that the slave responds with the appropriate data. Make sure the buffer is filled before the SPI is armed (*Nano_SPISlaveArm()*).

hDevice: This is the handle to the USB device.

Channel: This specifies which of the particular buffer to fill with data.

Channel	Meaning
NANO_SPI_CHANNEL_0	Fills the channel 0 buffer.
NANO_SPI_CHANNEL_1	Fills the channel 1 buffer.
NANO_SPI_CHANNEL_2	Fills the channel 2 buffer.
NANO_SPI_CHANNEL_3	Fills the channel 3 buffer.
. . .	. . .
NANO_SPI_CHANNEL_15	Fills the channel 15 buffer.
NANO_SPI_CHANNEL_16	Fills the channel 16 buffer.

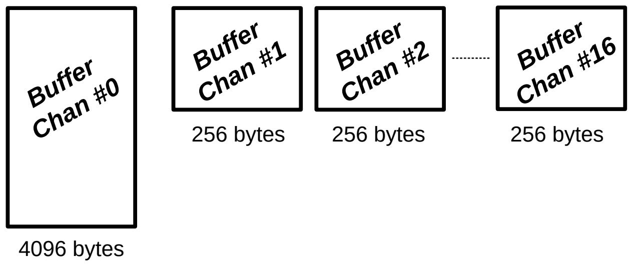
Length: Sets the amount of data to be filled in the buffer. Maximum is 4096 bytes for channel 0 and 256 bytes for channels 1-16.

**pOutBuffer*: This is a pointer to an array which should contain the data to be written to the buffer.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

13.3. Nano_SPISlaveBufferRead

NANO_RESULT **Nano_SPISlaveBufferRead (**
HANDLE *hDevice,*
BYTE *Channel,*
WORD *Length,*
BYTE **pInBuffer*
)



The ViperBoard SPI Slave has a buffer for each channel. Prior to the SPI transfer the buffer contains the data to be sent to the master. After the transaction then the buffer will contain the data written by the master. The buffer for channel 0 contains 4096 bytes. The buffers for channels 1-16 each contain 256 bytes.

This command allows one of the SPI Slave buffers to be read. This would be used for example after an SPI transaction.

hDevice: This is the handle to the USB device.

Channel: This specifies which of the particular buffers to retrieve data from.

Channel	Meaning
NANO_SPI_CHANNEL_0	Retrieve the channel 0 buffer.
NANO_SPI_CHANNEL_1	Retrieve the channel 1 buffer.
NANO_SPI_CHANNEL_2	Retrieve the channel 2 buffer.
NANO_SPI_CHANNEL_3	Retrieve the channel 3 buffer.
. . .	. . .
NANO_SPI_CHANNEL_15	Retrieve the channel 15 buffer.
NANO_SPI_CHANNEL_16	Retrieve the channel 16 buffer.

Length: Sets the amount of data to be read from the buffer. Maximum is 4096 bytes for channel 0 and 256 bytes for channels 1-16.

**pInBuffer*: This is pointer to an array which is filled with data from the buffer.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

14. Analogue Input Tasks

In this section the tasks for the Analogue Input interface are described in full detail.

14.1. Nano_ADCRead

```
NANO_RESULT     Nano_ADCRead (
                HANDLE    hDevice,
                BYTE      Channel,
                BYTE      *pValue
                )
```

This function allows reading of one of the 4 analogue inputs.

hDevice:	This is the handle to the USB device.
Channel:	This is the particular analogue channel to read. Selection can be 0, 1, 2 or 3.
**pValue*:	This is a pointer to be filled with the analogue to digital converted input data for the selected channel. Conversion is made with 8-bit resolution. 0x00 is minimum and 0xFF is maximum.

The function returns NANO_RESULT. NANO_SUCCESS is a success and NANO_XACTION_FAILURE is a failure.

15. Pre-defined Types and Constants

15.1. NANO_RESULT

A type is defined to describe different exit results from ViperBoard tasks.

```
typedef enum
{
        NANO_SUCCESS = 0,                  // Call was successful
        NANO_XACTION_FAILURE = 1,          // There was an error in the USB transaction
        NANO_HW_NOT_FOUND = 2,             // No hardware was found
        NANO_BAD_PARAMETER = 3,            // Bad or out of range parameters
        NANO_IIC_PROTOCOL_ERROR = 4        // There was an error discovered in the IIC transfer
                                           // (for example an ACK error)
} NANO_RESULT;
```

15.2. NANO_GPIO_CLOCK

A type is defined to describe different base clocks used in GPIOA functions.

```
typedef enum
{
        NANO_GPIO_CLK_1      = 0,          //  (1us   = 1MHz)
        NANO_GPIO_CLK_10     = 1,          //  (10us  = 100kHz)
        NANO_GPIO_CLK_100    = 2,          //  (100us = 10kHz)
        NANO_GPIO_CLK_1000   = 3,          // (1ms   = 1kHz)
        NANO_GPIO_CLK_10000  = 4,          //  (10ms  = 100Hz)
        NANO_GPIO_CLK_100000 = 5           // (100ms = 10Hz)

} NANO_GPIO_CLOCK;
```

15.3. NANO_I2C_FREQ

A type is defined to declare different I2C clock frequencies.

```
typedef enum
{
        NANO_I2C_FREQ_6MHZ   = 1,          //  6 MBit/s
        NANO_I2C_FREQ_3MHZ   = 2,          //  3 MBit/s
        NANO_I2C_FREQ_1MHZ   = 3,          //  1 MBit/s
        NANO_I2C_FREQ_FAST   = 4,          //  400 kbit/s
        NANO_I2C_FREQ_200KHz = 5,          //  200 kbit/s
        NANO_I2C_FREQ_STD    = 6,          //  100 kbit/s
        NANO_I2C_FREQ_10KHZ  = 7           //  10 kbit/s

} NANO_I2C_FREQ;
```

15.4. NANO_I2C_TRANSFER

A type is defined to distinguish between different types of I2C transfer type.

```
typedef enum
{
        NANO_I2C_WRITE_TRANSFER      = 0,     // A single I2C write transfer
        NANO_I2C_READ_TRANSFER       = 1,     // A single I2C read transfer
        NANO_IC_WRITEREAD_TRANSFER   = 2,     // A write then read I2C transfer
        NANO_IC_READWRITE_TRANSFER   = 3,     // A read then write I2C transfer
        NANO_IC_WRITEWRITE_TRANSFER  = 4,     // A back-back write I2C transfer
        NANO_IC_READREAD_TRANSFER    = 5      // A back-back read I2C transfer

} NANO_I2C_TRANSFER;
```

15.5. NANO_SPI_FREQ

A type is defined to declare different SPI clock frequencies.

```
  typedef enum
  {
        NANO_SPI_FREQ_12MHZ  = 0,     //  12 MBit/s
        NANO_SPI_FREQ_6MHZ   = 1,     //   6 MBit/s
        NANO_SPI_FREQ_3MHZ   = 2,     //   3 MBit/s
        NANO_SPI_FREQ_1MHZ   = 3,     //   1 MBit/s
        NANO_SPI_FREQ_400KHZ = 4,     // 400 kbit/s
        NANO_SPI_FREQ_200KHZ = 5,     // 200 kbit/s
        NANO_SPI_FREQ_100KHZ = 6,     // 100 kbit/s
        NANO_SPI_FREQ_10KHZ  = 7      //  10 kbit/s

  } NANO_SPI_FREQ;
```

15.6. NANO_SPI_CHANNEL

A type is defined to declare which pin is used as the SPI chip select.

```
typedef enum
{
        NANO_SPI_CHANNEL_0  = 0,        // Master or Slave SPI connector
        NANO_SPI_CHANNEL_1  = 1,        // Corresponds to GPIO_A_00
        NANO_SPI_CHANNEL_2  = 2,        // Corresponds to GPIO_A_01
        NANO_SPI_CHANNEL_3  = 3,        // Corresponds to GPIO_A_02
        NANO_SPI_CHANNEL_4  = 4,        // Corresponds to GPIO_A_03
        NANO_SPI_CHANNEL_5  = 5,        // Corresponds to GPIO_A_04
        NANO_SPI_CHANNEL_6  = 6,        // Corresponds to GPIO_A_05
        NANO_SPI_CHANNEL_7  = 7,        // Corresponds to GPIO_A_06
        NANO_SPI_CHANNEL_8  = 8,        // Corresponds to GPIO_A_07
        NANO_SPI_CHANNEL_9  = 9,        // Corresponds to GPIO_A_08
        NANO_SPI_CHANNEL_10 = 10,       // Corresponds to GPIO_A_09
        NANO_SPI_CHANNEL_11 = 11,       // Corresponds to GPIO_A_10
        NANO_SPI_CHANNEL_12 = 12,       // Corresponds to GPIO_A_11
        NANO_SPI_CHANNEL_13 = 13,       // Corresponds to GPIO_A_12
        NANO_SPI_CHANNEL_14 = 14,       // Corresponds to GPIO_A_13
        NANO_SPI_CHANNEL_15 = 15,       // Corresponds to GPIO_A_14
        NANO_SPI_CHANNEL_16 = 16        // Corresponds to GPIO_A_15

} NANO_SPI_CHANNEL;
```

15.7. NANO_PIN_MODE

A type is defined to declare a mode setting for a particular Port A GPIO.

```
typedef enum
{
        GPIO_MODE = 0,                  // Pin used as a GPIO
        SPI_SLAVE_MODE = 1,             // Pin used as an SPI slave chip select input
        SPI_MASTER_MODE = 2             // Pin used as an SPI master chip select output

} NANO_PIN_MODE;
```

15.8. DEV_LIST

A type is declared to contain a list of all I2C addresses for devices connected to the I2C bus. If the element of the array is TRUE then the index for this element is the I2C address for a connected device – i.e. if List[0x50] is TRUE, then an I2C device with device ID of 0x50 is connected. If FALSE then a chip is not connected with that device ID.

```
typedef struct
{
        BOOL List[128];

} DEV_LIST;
```