NXP Semiconductors
User Guide

Document identifier: LIBUSBSIOUG
Rev. 0, 27 July 2021

USBSIO Library User's Guide

Contents

Chapter 1
Introduction

The USB serial I/O library (USBSIO or LIBUSBSIO) is a generic API provided to PC applications programmers to develop applications communicating with target NXP microcontroller devices using SPI, I2C, and GPIO over a USB bridge.

The library converts its API calls into USB messages which are transferred to a USBSIO *bridge device*, which in turn communicates with the target microcontroller using the physical communication interface. The bridge is implemented by LPCLink2 or MCULink debug probes used either as a standalone interface or assembled directly on an evaluation board and connected to a target microcontroller.

The USBSIO library uses the USB HID class as a transport mechanism, which is widely supported by all common operating systems. It is provided as a static or dynamic binary library for Microsoft Windows, macOS, and Linux systems, and it is easy to use in C/C++ applications. The publicly available, open source *hidapi* low-level library is used internally by USBSIO, because it simplifies the use of the HID communication class across all supported operating systems. You can find more information about the hidapi library at https://github.com/signal11/hidapi.

The USBSIO functionality is just an optional additional feature of the LPCLink2 and MCULink Pro devices. Their primary function is to provide a CMSIS-DAP debugging interface to the target microcontroller and a virtual serial communication using the USB VCOM class. The SEGGER J-Link firmware option, which is also available for these debug probes, does not support the USB bridge function for SPI, I2C, and GPIO. All communication options are displayed in Figure 1. All USB classes are implemented as a composite device and accessed from a PC host over a single USB cable.

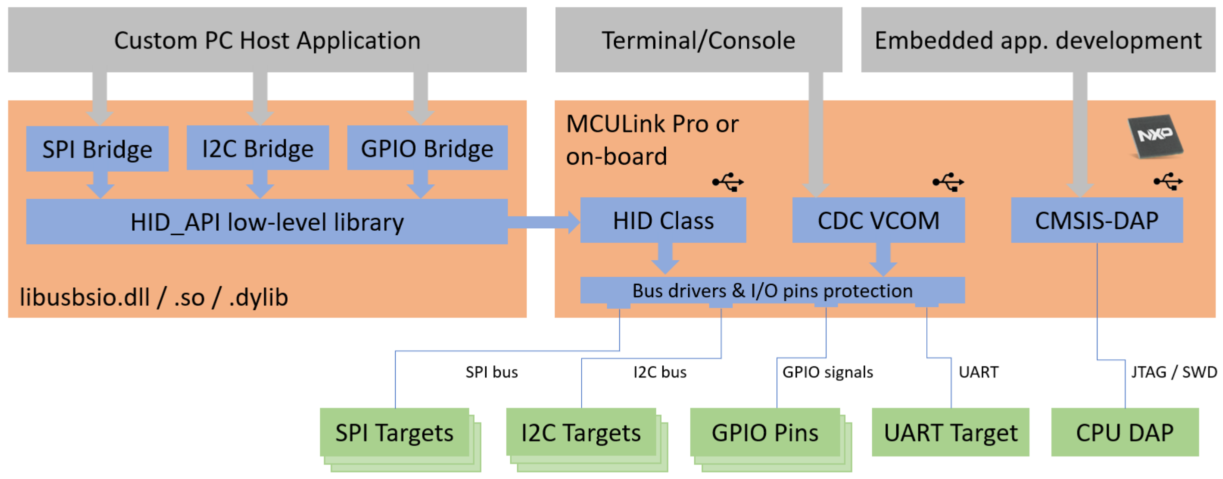

Figure 1. Communication options

1.1 History and backward compatibility

The USBSIO library v2.1 (and later) is a successor of an older implementation named LPCUSBSIO, which accompanied the LPCLink2 interface drivers. The new library API is backward compatible for the whole SPI and GPIO functionality. The key I2C communication API is also backward compatible, except for some special transfer options that were dropped in the latest version.

See more details about the removed functionality in the I2C API reference.

Even though the library v2.1 is renamed to LIBUSBSIO, the API naming remains backward compatible and still uses the “LPCUSBSIO_” prefix.

1.2 Installation

The library package is available as a ZIP file which includes the binary library files built for all supported platforms, header files needed to access the library API in C/C++ applications, example code, and user documentation. Unzip the package to an arbitrary location and access the files there.

1.3 Python wrapper

The Python wrapper class is provided along with the LIBUSBSIO library and enables all library features to be used also from the Python scripting language. The Python installable module is available at the https://PyPI.org site. The package contains the binary libraries for all supported operating systems and the Python wrapper class code. Use the "pip" package manager to download and install the package as follows:

```
pip install libusbsio
```

Using the USBSIO library from Python scripts and applications is a good "platform independent" alternative to developing a C/C++ application and linking it with the library in a native mode.

Chapter 2
Developing serial IO application with USBSIO library

This section describes the use of the USBSIO library in native C/C++ applications.

2.1 Prerequisites

- A host system supported by the USBSIO library.
- A target NXP microcontroller board with onboard LPCLink2 or MCULink interfaces. Alternatively, a standalone MCULink Pro device can be used and interconnected with the target microcontroller board.
- The LIBUSBSIO library package.
- The C/C++ compiler toolchain or IDE for the host system.

2.2 Setting up project environment

This section uses the simple C test application code included in the installation package in the test/testapp directory. The test application demonstrates how to use the library API in a custom code and how to link the application executable with the library, either statically or dynamically.

The test application is available as:

- Microsoft Visual Studio project for Windows builds
- Common *Makefile* for Linux and macOS builds

Use the library in a custom application as follows:

- Include the libusbsio.h header file in the application C or C++ source files.
- Link the application with the library. The official binary libraries are in the bin directory.

In the Windows operating systems:

- Link the application statically with the libusbsio.lib file available in the bin/Win32 or bin/x64 directories. Use bin/$(Platform) in project settings when building multi-platform applications.
- The executable can also be built using the libusbsio.dll.lib file, which is a wrapper library that automatically loads the libusbsio.dll file after startup. In this case, build the final executable with the LPCUSBSIO_IMPORTS macro defined and distribute the DLL along with your application.
- The testApp example project has the Debug and Release targets to link with the DLL library wrapper. Use the DebugS and ReleaseS targets to link with the static library.

On Linux operating systems:

- Link the application statically with the libusbsio.a file available in the bin/linux directory. This is a 64-bit build tested on the Ubuntu 20.04 operating system. The bin/linux32 directory contains a 32-bit build tested with Ubuntu 16.04.
- Add the pkg-config libusb-1.0 libudev –libs option to the LIBS makefile variable when linking the final application executable, so that the libusb and libudev libraries are also linked.
- If the libusb library is missing, install it using the sudo apt-get install -y libusb-1.0-0 command.
- If the libudev library is missing, install it using the sudo apt-get install -y libudev-dev command.
- The executable can also be built using the libusbsio.so dynamic library.

On macOS systems:

- Link the application statically with the libusbsio.a file in the bin/osx directory. This is a 64-bit build tested on macOS 11.3.

- Add the **-framework IOKit -framework CoreFoundation** option to the LIBS makefile variable when linking the final application executable.
- The executable can also be built using **libusbsio.dylib** dynamic library.

2.3 Obtaining Serial IO device handle

The enumeration of all USBSIO bridge devices is combined into a single LPCUSBSIO_GetNumPorts library call. This function performs an enumeration and returns the number of bridge devices found.

- All applications using the library should call this routine before using any other library API.
- This function takes the Vendor ID and Product ID (VID, PID pair) of the bridge devices to enumerate.
- LPCLink2 and MCULink use different PID identifiers. Refer to a code section below for the LPCUSBSIO_VID value (0x1FC9) and the other PID constants defined (0x90 and 0x143).
- Multiple devices of the same VID/PID class may be connected and they are all enumerated.
- After the debug probe has enumerated and the function call returns a non-zero count, specify a zero-based index of the selected bridge device in a subsequent call to LPCUSBSIO_Open.
- A device which is already open is not enumerated if LPCUSBSIO_GetNumPorts is called again.
- The SIO handle returned by LPCUSBSIO_Open is used by all consequent calls targeted to the device.
- Call LPCUSBSIO_GetVersion to obtain the string description of the library and the target bridge firmware versions. For example:

```
NXP LIBUSBSIO v2.1 (Mar 30 2021 14:41:19)/FW 2.0 (May 18 2021 07:44:09)
```

- The SIO device handle should be closed by calling LPCUSBSIO_Close when it is no longer required.

The following code shows the use of basic library calls:

```
#include "lpcusbsio.h"
#include <stdio.h>
#define LPCUSBSIO_VID 0x1FC9
#define LPCLINKSIO_PID 0x0090
#define MCULINKSIO_PID 0x0143
LPC_HANDLE open_usbsio(void)
{
    LPC_HANDLE hSIOPort = NULL;
    int res;
    if((res = LPCUSBSIO_GetNumPorts(LPCUSBSIO_VID, LPCUSBSIO_PID)) > 0)
    {
        printf("Total LPCLink2 devices: %d\r\n", res);
    }
    else if((res = LPCUSBSIO_GetNumPorts(LPCUSBSIO_VID, MCULINKSIO_PID)) > 0)
    {
        printf("Total MCULink devices: %d\r\n", res);
    }
    else
    {
        printf("No USBSIO bridge device found\r\n");
    }
    if (res > 0)
    {
        /* open device at index 0 */
        printf("Using device #0\r\n");
        hSIOPort = LPCUSBSIO_Open(0);
        printf("Device version: %s \n ", LPCUSBSIO_GetVersion(hSIOPort));
    }
    return hSIOPort;
}
```

2.4 Accessing the SPI, I2C, and GPIO ports

When a valid SIO handle is returned by the LPCUSBSIO_Open call, it can be used to determine the number of SPI, I2C, and GPIO ports supported by the bridge interface.

- Call LPCUSBSIO_GetNumI2CPorts, LPCUSBSIO_GetNumSPIPorts, and LPCUSBSIO_GetNumGPIOPorts, respectively.
- The GPIO subsystem API can be accessed directly using the SIO library handle returned by LPCUSBSIO_Open.
- The I2C and SPI ports must be open by calling I2C_Open and SPI_Open functions. Handles returned by these calls are then used in subsequent I2C or SPI API calls.
- Call LPCUSBSIO_GetMaxDataSize to determine the maximum SPI or I2C data length used in a single transaction. The typical value supported by LPCLink2 and MCU-Link bridge is 1024.

2.5 Initializing and obtaining I2C port handle

When the SIO handle is obtained and the number of I2C ports available on the bridge device is known, then individual I2C ports can be opened and I2C transfers can be initiated.

To obtain an I2C port handle, the application should call I2C_Open with the SIO device handle, port number, and port configuration as the parameters. This also initializes the corresponding I2C interface of the bridge device. The port configuration only contains the I2C clock speed. Use one of the standard 100 kHz or 400 kHz speeds.

When a valid (non-NULL) I2C port handle is obtained, the port is ready for data transfers. The handle should be closed by calling I2C_Close when it is no longer required by the application.

The following code snippet shows the I2C initialization steps.

```
void open_i2c(LPC_HANDLE hSIOPort, int port)
{
    int err_code = LPCUSBSIO_OK;
    LPC_HANDLE hI2CPort = NULL;
    I2C_PORTCONFIG_T cfgParam;
    /* Init the I2C port for standard speed communication */
    cfgParam.ClockRate = I2C_CLOCK_STANDARD_MODE;
    cfgParam.Options = 0;
    /* open I2C0 port */
    hI2CPort = I2C_Open(hSIOPort, &cfgParam, 0);
    if (hI2CPort != NULL)
    {
        printf("I2C port opened successfully\n", res);
        /* communicate over I2C ... */
        /* close the port */
        I2C_Close(hI2CPort);
    }
}
```

2.6 Reading and writing to I2C target device

The library provides two types of APIs for transferring data to and from I2C target devices connected to the target MCULink bridge.

- Unidirectional: This group contains independent I2C_DeviceWrite and I2C_DeviceRead routines.
- Bidirectional: A single API I2C_FastXfer routine performs write and read-after-write transfers.

Backward compatibility note: Older library versions (LPCUSBSIO) supported communication options flags enabling override of the standard behavior of I2C transactions, for example preventing the generation of the START or STOP signals. These flags are not supported by the MCULink firmware and ignored if used in the USBSIO library calls.

2.6.1 Unidirectional I2C transactions

The maximum data size supported by the transfer routines depends on the transfer size supported by the bridge firmware which can be found out by calling LPCUSBSIO_GetMaxDataSize. Since this is typically a large number (1024 bytes) and the HID report size is 64 bytes, larger transfers are split into multiple USB frames. When the data is transferred to the bridge device, it performs the I2C transfer and returns the result code and any data read. The response may be also split into multiple USB frames for large data. Data integrity is assured by the USBSIO library.

There are two API calls used for unidirectional I2C transactions:

I2C_DeviceWrite()

- I2C_DeviceWrite performs a single I2C write operation which consists of:
  - START signal
  - ADDRESS byte with R/W bit clear denoting the write operation
  - DATA payload
  - STOP signal
- The I2C_DeviceWrite expects the ADDRESS and DATA bytes to be acknowledged by the target device.
- Special options flags can be used to alter the standard behavior and for example, omit the ADDRESS byte. The flags are described in the reference section, but it is not typical to use any of them. Also, some options flags are only supported by an older LPCLink2 device and not by the MCULink.

I2C_DeviceRead()

- I2C_DeviceRead performs a single I2C read operation which consists of:
  - START signal
  - ADDRESS byte with the R/W bit set, denoting the read operation
  - Reading the DATA payload and sending the acknowledgment for each byte
  - STOP signal
- Special options flags can be used to alter the standard behavior and for example, omit the ADDRESS byte. The flags are described in the reference section, but it is not typical to use any of them. Some options flags are only supported by the LPCLink2 and not the MCULink.

2.6.2 Bidirectional transfer routines

Most of the I2C read transactions made with real I2C devices are typically preceded with a write operation defining a register index or other specification of data to be read. In this scenario, the STOP signal is not generated between the write and read operations. The repeated START is generated instead.

Such operation would be difficult to achieve with unidirectional transfer routines described above. It would be necessary to use special option flags to prevent a STOP signal after a write and there would also be an unavoidable round-trip delay between the write and read parts of the operation. The I2C bus would be held busy for a long time which could cause performance issues in multi-master scenarios.

The I2C_FastXfer API call is recommended to perform such read-after-write transactions, but it can also be used for write-only and read-only transactions.

I2C_FastXfer()

- I2C_FastXfer performs a combined read-after-write I2C transaction which consists of:
  - START signal

  - ADDRESS byte with R/W bit clear denoting the write operation
  - DATA payload
  - Repeated START signal
  - ADDRESS byte with the R/W bit set, denoting the read operation
  - Reading the DATA payload and sending the acknowledgment for each byte
  - STOP signal
- The write portion can be skipped if no data are provided and zero is passed as write length.
- The read portion can be skipped if zero is passed to the expected read length.
- Special options flags can be used to alter the standard behavior of ACK and NAK handling, but it is not typical to use any of them. Also, some options flags are only supported by the LPCLink2 device but not by MCULink.

The following code snippet shows how to use the I2C_FastXfer call.

```
int write_read_i2c(LPC_HANDLE hI2CPort, uint8_t* txData, uint16_t txLen,
uint8_t* rxBuff, uint16_t rxLen)
{
  I2C_FAST_XFER_T xfer;
  int result = LPCUSBSIO_ERR_BAD_HANDLE;
  if (hI2CPort != NULL)
  {
    xfer.slaveAddr = 0x10; /* 7-bit I2C target device address */
    xfer.txBuff = txData; /* Pointer to bytes to be transmitted */
    xfer.txSz = txLen; /* Number of bytes to transmit */
    xfer.rxBuff = rxBuff; /* Memory where bytes received are to be stored */
    xfer.txSz = rxLen; /* Number of bytes to receive */
    xfer.options = 0; /* No extra options needed */
    /* Execute transfer */
    result = I2C_FastXfer(hI2CPort, &xfer);
    printf("I2C transfer returned %d\n", result);
  }
  return result;
}
```

2.7 Initializing and obtaining SPI port handle

When the SIO handle is obtained and the number of SPI ports available on the MCULink bridge is known, then individual SPI ports can be opened and SPI transfers can be initiated.

To obtain an SPI port handle, the application should call SPI_Open with the SIO device handle, port number, and port configuration as the parameters. This also initializes the corresponding SPI interface of the bridge device. The port configuration contains the SPI clock speed, data size in bits, clock phase, polarity, and data-to-clock delay parameters.

Once a valid (non-NULL) SPI port handle is obtained, the port is ready for data transfers. The handle should be closed by calling SPI_Close when it is no longer required by the application.

The following code snippet shows the SPI initialization steps.

```
void open_spi(LPC_HANDLE hSIOPort, int port)
{
      int err_code = LPCUSBSIO_OK;
      LPC_HANDLE hSPIPort = NULL;
      SPI_PORTCONFIG_T cfgParam;
      /* Init the SPI port for standard speed communication */
      cfgParam.busSpeed = 1000000;
      cfgParam.Options = HID_SPI_CONFIG_OPTION_DATA_SIZE_8 | /* 8 bits */
```

```
       HID_SPI_CONFIG_OPTION_POL_0 | /* CPOL=0 */
       HID_SPI_CONFIG_OPTION_PHA_0 | /* CPHA=0 */
       HID_SPI_CONFIG_OPTION_PRE_DELAY(100); /* 100 us */
       /* open SPI0 port */
       hSPIPort = SPI_Open(hSIOPort, &cfgParam, 0);
       if (hSPIPort != NULL)
       {
             printf("SPI port opened successfully\n", res);
             /* communicate over SPI ... */
             /* close the port */
             SPI_Close(hSPIPort);
       }
}
```

2.8 Data transfer with an SPI target device

Since the SPI is a full duplex communication protocol, transmission and reception happen at the same time and both the transmit and receive lengths are the same. The library provides a single transfer API (SPI_Transfer) to transfer data to and from SPI target devices connected to the target LPC controller.

The SPI SSEL signal which selects which target SPI device is addressed is specified with the LPCLink2 device. A GPIO port and pin numbers of the SSEL signal are used in the transfer call as parameters. The MCU Link supports a single target SPI device, so the SSEL0 signal is driven regardless of the port pin parameter values.

The maximum data transfer size supported by the transfer routines depends on the transfer size supported by the bridge firmware which is found out by calling LPCUSBSIO_GetMaxDataSize.

SPI_Transfer()

The following are the SPI_Transfer library function details:

- It performs a single bidirectional transfer over an SPI bus.
- The function accepts the open SPI port handle and a single SPI_XFER_T structure pointer as parameters.
- The transfer structure contains the SSEL target device selection, data length specification, and a pair of data buffers which must be at least “length” bytes long.
- The data of the transmit buffer are sent. The receive buffer is filled with the data received. The function returns the number of bytes transferred.

The following code snippet is the SPI_Transfer call:

```
int write_read_spi(LPC_HANDLE hSPIPort, uint8_t* txData, uint8_t* rxBuff, uint16_t len)
{
        SPI_XFER_T xfer;
        int result = LPCUSBSIO_ERR_BAD_HANDLE;
        if (hSPIPort != NULL)
        {
                xfer.length = len; /* Number of bytes to transmit and receive */
                xfer.options = 0; /* No extra options currently supported */
                xfer.device = LPCUSBSIO_GEN_SPI_DEVICE_NUM(0, 1); /* SPI SSEL port and pin*/
                xfer.txBuff = txData; /* Pointer to bytes to be transmitted */
                xfer.rxBuff = rxBuff; /* Memory where bytes received are to be stored */
                /* Execute transfer */
                result = SPI_Transfer(hSPIPort, &xfer);
                printf("SPI transfer returned %d\n", result);
        }
        return result;
}
```

2.9 Accessing GPIO ports and pins

The USBSIO library provides an API to control GPIO signals of the SIO bridge device either as individual pins or as an entire port group. The ports and pins are identified with respect to the central CPU of the bridge device.

- The LPCLink2 device available on NXP evaluation boards provides one pin at port=0 and pin=15.
- The MCU Link available on NXP evaluation boards provides up to six pins at port=1, pins=[1, 7, 9, 20, 21, 31]. The number of pins provided depends on the particular evaluation board design.
- The MCU Link Pro standalone interface does not provide any GPIO pins.
- Some of the pins used for I2C and SPI communication may also be redefined for a direct GPIO control by calling GPIO_ConfigIOPin() with a proper argument:
  - Mode value of 0x100 configures a pin for GPIO push-pull function. Use it for SPI pins.
  - Mode value of 0x300 configures a pin for GPIO open-drain function. Use it for I2C pins.
- Hardware bus drivers and optional level shifters define the signal direction. There is a possible signal collision with other SPIs and I2Cs that are on the board.
  - The port and pin identification of communication pins can be found in the related hardware documentation and schematics.
  - SPI.MISO may only be redefined to a GPIO input.
  - SPI.SCK, SSEL, and MOSI may only be redefined as GPIO outputs.
  - I2C.SDA, and SCL may be used as an input or an open-drain output. A pull-up resistor is assembled on the board.
  - When the SPI or I2C communication takes place, the GPIO functionality is de-activated and all related signals return to the original operation mode.

Use LPCUSBSIO_GetNumGPIOPorts to determine whether the GPIO functionality is available. When enabled, the returned value is typically higher than 1, because it reflects the total number of GPIO ports of the main bridge CPU. Each port consists of 32 pins. However, only some ports and some pins are usable as GPIO, as mentioned above.

2.10 Error propagation

Most APIs return zero or a positive number on success and negative numbers in the case of an error. The error types are defined by the LPCUSBSIO_ERR_T enumeration type. The library also provides the LPCUSBSIO_GetLastError and LPCUSBSIO_Error routines to obtain the last error as a numeric code or as a unicode string.

2.11 Deploying LPCUSBSIO applications

2.11.1 Microsoft Windows OS

The deployment depends on what library is used to build the final application executable. If the application was linked with a static **libusbsio.lib** library, the executable contains all required code and it can be deployed standalone. If **libusbsio.dll.lib** or a different method of dynamic linking is used, then the **libusbsio.dll** file should be always included with the final executable in the distribution and it should be in the same directory as the executable.

2.11.2 Linux OS and macOS

Both static and dynamic libraries are also available for Linux OS and macOS applications. Static linking to **libusbsio.a** is the preferred way of using the library.

The dynamic linking library is provided mainly for compatibility with the Python wrapper.

2.11.3 Python

A Python application may leverage the pip package manager to resolve the dependency on the libusbsio module and install all required files. The Python module contains dynamic linking libraries for all platforms and it loads the correct one automatically as soon as the first instance of the LIBUSBSIO class is created.

The LIBUSBSIO class enables Python scripts to use the whole functionality of the USBSIO library. The class instance wraps the SIO device handle, so its methods do not need to specify the handle again. The class destructor closes the handle automatically. Similarly, the I2C_Open and SPI_Open methods return Python class instances, which also wrap the underlying port handles. Many class methods use keyword arguments and arguments with default values, which makes the final Python code well readable.

Chapter 3 Library API reference

3.1 General library functions

This section describes the general library functions.

- **int32_t LPCUSBSIO_GetNumPorts (uint32_t vid, uint32_t pid)**

  gets the number of USBSIO ports implemented by the connected bridge devices.
- **LPC_HANDLE LPCUSBSIO_Open (uint32_t index)**

  opens the indexed Serial IO port.
- **int32_t LPCUSBSIO_Close (LPC_HANDLE hUsbSio)**

  closes the Serial IO port.
- **const char* LPCUSBSIO_GetVersion (LPC_HANDLE hUsbSio)**

  gets the version string of the USBSIO library.
- **uint32_t LPCUSBSIO_GetNumI2CPorts (LPC_HANDLE hUsbSio)**

  returns the number of I2C ports supported by the Serial IO device.
- **uint32_t LPCUSBSIO_GetNumSPIPorts (LPC_HANDLE hUsbSio)**

  returns the number of SPI ports supported by the Serial IO device.
- **uint32_t LPCUSBSIO_GetNumGPIOPorts (LPC_HANDLE hUsbSio)**

  returns the number of GPIO ports supported by the Serial IO device.
- **uint32_t LPCUSBSIO_GetMaxDataSize (LPC_HANDLE hUsbSio)**

  returns the maximum number of bytes supported for I2C/SPI transfers by the Serial IO device.
- **const wchar_t* LPCUSBSIO_Error (LPC_HANDLE hUsbSio)**

  gets a string describing the last error that occurred.
- **int32_t LPCUSBSIO_GetLastError (void)**

  returns the last error seen by the library.

3.1.1 LPCUSBSIO_GetNumPorts

Prototype

```
int32_t LPCUSBSIO_GetNumPorts (uint32_t vid, uint32_t pid)
```

Parameters

- **vid** … USB Vendor ID (VID)
- **pid** … USB Product ID (PID)

Return

The number of SIO ports implemented by one or more USB SIO bridge devices specified by the VID and PID identification values.

Remarks

This function enumerates the USB bus and searches for the USBSIO devices specified by the VID and PID pairs. The following VID and PID pairs are currently supported:

- **LPCLink2**: VID=0x1FC9, PID=0x0090
- **MCULink**: VID=0x1FC9, PID=0x0143

This function also performs an internal device list initialization, so it must be called before opening any SIO port, even if the VID, PID, and SIO device indexes are known.

3.1.2 LPCUSBSIO_Open

Prototype

```
LPC_HANDLE LPCUSBSIO_Open (uint32_t index)
```

Parameters

- **index** … index of SIO port device to open

Return

The handle of the SIO port.

Remarks

Call this function after enumerating the bridge devices by the LPCUSBSIO_GetNumPorts function. This function opens the specified SIO port by a numeric index.

3.1.3 LPCUSBSIO_Close

Prototype

```
int32_t LPCUSBSIO_Close (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

Zero if successful, error codes are defined in the LPCUSBSIO_ERR_T enumeration.

Remarks

Always close a serial IO port handle when it is no longer needed.

3.1.4 LPCUSBSIO_GetVersion

Prototype

```
const char* LPCUSBSIO_GetVersion (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

Version string which consists of the USBSIO library version and a bridge device firmware version separated by a slash.

Remarks

Get the version string of the USBSIO library and the bridge device firmware.

3.1.5 LPCUSBSIO_GetNumI2CPorts

Prototype

```
uint32_t LPCUSBSIO_GetNumI2CPorts (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

The number of available I2C ports.

Remarks

Use the I2C_Open call to open one of the available I2C ports.

3.1.6 LPCUSBSIO_GetNumSPIPorts

Prototype

```
uint32_t LPCUSBSIO_GetNumSPIPorts (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

The number of the available SPI ports.

Remarks

Use the SPI_Open call to open one of the available SPI ports.

3.1.7 LPCUSBSIO_GetNumGPIOPorts

Prototype

```
uint32_t LPCUSBSIO_GetNumGPIOPorts (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

The number of the available GPIO ports.

Remarks

Returns the number of GPIO ports supported by the Serial IO device. Not all GPIO ports are available for the input or output on the bridge device pins. The returned value reflects the total number of GPIO ports, but there are typically just a few pins of a single GPIO port available externally.See Accessing GPIO Ports and Pins for more information about the GPIO ports and pins.

3.1.8 LPCUSBSIO_GetMaxDataSize

Prototype

```
uint32_t LPCUSBSIO_GetMaxDataSize (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

The maximum size of one SPI or I2C data transfer made by a single library call.

Remarks

The return value is 1024 bytes in all currently known implementations. Because the USB HID report size is 64 bytes, larger transfers are split into multiple USB frames. When the data is transferred to the bridge device, it performs the SPI or I2C transfer and returns the result code and any data read. The response may be also split into multiple USB frames for large data. The data integrity is assured by the USBSIO library.

3.1.9 LPCUSBSIO_Error

Prototype

```
const wchar_t* LPCUSBSIO_Error (LPC_HANDLE hUsbSio)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

A wide character string with the last error description.

Remarks

An empty string is returned when no error has occurred.

3.1.10 LPCUSBSIO_GetLastError

Prototype

```
LPCUSBSIO_API int32_t LPCUSBSIO_GetLastError (void)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open

Return

Zero if the last operation was successful. All error codes are defined in the LPCUSBSIO_ERR_T enumeration.

3.2 I2C communication functions

This section describes the functions used for the I2C communication.

- **LPC_HANDLE I2C_Open (LPC_HANDLE hUsbSio, I2C_PORTCONFIG_T config, uint8_t portNum)**

  initializes an I2C port.
- **int32_t I2C_Close (LPC_HANDLE hI2C)**

  closes an I2C port.
- **int32_t I2C_Reset (LPC_HANDLE hI2C)**

  resets an I2C controller.
- **int32_t I2C_DeviceRead (LPC_HANDLE hI2C, uint8_t deviceAddress, uint8_t *buffer, uint16_t sizeToTransfer, uint8_t options)**

  reads from an addressed I2C target device.
- **int32_t I2C_DeviceWrite (LPC_HANDLE hI2C, uint8_t deviceAddress, uint8_t *buffer,**

  uint16_t sizeToTransfer, uint8_t options)

  writes to the addressed I2C target device.
- **int32_t I2C_FastXfer (LPC_HANDLE hI2C, I2C_FAST_XFER_T *xfer)**

  transmits and receives the data in the I2C master mode.

3.2.1 I2C_Open

Prototype

```
LPC_HANDLE I2C_Open (LPC_HANDLE hUsbSio, I2C_PORTCONFIG_T config, uint8_t portNum)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **config** … Pointer to the I2C configuration structure
- **portNum** … I2C port number in the limit defined by LPCUSBSIO_GetNumI2CPorts

I2C_PORTCONFIG_T structure

- **ClockRate** … I2C bus clock rate. Use one of the constants pre-defined in the I2C_ClockRate_t enumeration:
  - **I2C_CLOCK_STANDARD_MODE …** 100 kbit/s
  - **I2C_CLOCK_FAST_MODE** … 400 kbit/s
  - **I2C_CLOCK_FAST_MODE_PLUS** … 1 Mbit/s
- **Options**… additional configuration options. Currently unused.

Return

I2C port handle when successful. NULL in case of an error.

Remarks

The handle returned by this function is used in all subsequent calls to any I2C functions when accessing this I2C port.

3.2.2 I2C_Close

Prototype

```
int32_t I2C_Close (LPC_HANDLE hI2C)
```

Parameters

- **hI2C** … I2C port handle returned by I2C_Open

Return

Zero if successful. The error codes are defined in the LPCUSBSIO_ERR_T enumeration.

Remarks

Always close an I2C port handle when it is no longer needed

3.2.3 I2C_Reset

Prototype

```
int32_t I2C_Reset (LPC_HANDLE hI2C)
```

Parameters

- **hI2C** … I2C port handle returned by I2C_Open

Return

Zero if successful. The error codes are defined in the LPCUSBSIO_ERR_T enumeration.

Remarks

Reset the I2C controller. Use this function when the communication gets stuck or when the target nodes become unresponsive.

3.2.4 I2C_DeviceRead

Prototype

```
int32_t I2C_DeviceRead (LPC_HANDLE hI2C, uint8_t deviceAddress, uint8_t *buffer,
uint16_t sizeToTransfer, uint8_t options)
```

Parameters

- **hI2C** … I2C port handle returned by I2C_Open
- **deviceAddress** … 7-bit value of the target device address
- **buffer** … pointer to the memory buffer that receives the data
- **sizeToTransfer** … number of bytes to read
- **options** … additional option flags

Return

The number of bytes read from the I2C target device.

Remarks

Perform a uni-directional I2C read operation. For more information about uni-directional and bidirectional I2C operations, see Reading and Writing to I2C slave device. The last version of the USBSIO library does not support the communication options.

3.2.5 I2C_DeviceWrite

Prototype

```
int32_t I2C_DeviceWrite (LPC_HANDLE hI2C, uint8_t deviceAddress, uint8_t *buffer,
uint16_t sizeToTransfer, uint8_t options)
```

Parameters

- **hI2C** … I2C port handle returned by I2C_Open
- **deviceAddress** … 7-bit value of the target device address
- **buffer** … pointer to the data to be transmitted
- **sizeToTransfer** … number of bytes to write
- **options** … additional option flags

Return

Number of bytes written to the I2C target device.

Remarks

Perform a uni-directional I2C write operation. For more information about uni-directional and bidirectional I2C operations, see Reading and Writing to I2C slave device. The last version of the USBSIO library does not support the communication options.

3.2.6 I2C_FastXfer

Prototype

```
int32_t I2C_FastXfer (LPC_HANDLE hI2C, I2C_FAST_XFER_T *xfer)
```

Parameters

- **hI2C** … I2C port handle returned by I2C_Open
- **xfer** … the transfer structure containing the target address memory buffer pointers and sizes

I2C_FAST_XFER_T structure

- **slaveAddr** … 7-bit value of the target device address
- **txSz** … number of bytes prepared in the **txBuff** buffer to write to the I2C target device
- **rxSz** … size of the **rxBuff** buffer to receive the characters read from the I2C target device
- **txBuff** … bytes to write
- **rxBuff** … receive data buffer

- **options** … additional transfer option flags, currently unused

Return

Negative status code in case of a failure.

Remarks

Perform a bidirectional I2C write and read transaction without releasing the I2C bus. For more information about uni-directional and bidirectional I2C operations, see Reading and Writing to I2C slave device. The last version of the USBSIO library does not support the communication options.

3.3 SPI communication functions

This section describes the functions used for the I2C communication.

- **LPC_HANDLE SPI_Open (LPC_HANDLE hUsbSio, SPI_PORTCONFIG_T *config, uint8_t portNum)**

  initializes an SPI port.
- **int32_t SPI_Close (LPC_HANDLE hSPI)**

  closes an SPI port.
- **int32_t SPI_Reset (LPC_HANDLE hSPI)**

  resets an SPI controller.
- **int32_t SPI_Transfer (LPC_HANDLE hSPI, SPI_XFER_T *xfer)**

  transmits and receives the data in the SPI master mode.

3.3.1 SPI_Open

Prototype

```
LPC_HANDLE SPI_Open (LPC_HANDLE hUsbSio, SPI_PORTCONFIG_T *config, uint8_t portNum)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **config** … pointer to the SPI configuration structure
- **portNum** … SPI port number in the limit defined by LPCUSBSIO_GetNumSPIPorts

SPI_PORTCONFIG_T structure

- **busSpeed** … SPI clock rate. Use an unsigned long integer specifying the clock frequency. A typical value is 1000000UL for 1 Mbit/s.
- **Options**… additional configuration options. Use a bitwise OR combination of the following constants:
  - **SPI_CONFIG_OPTION_POL_0 / SPI_CONFIG_OPTION_POL_1** … clock polarity low/high
  - **SPI_CONFIG_OPTION_PHA_0 / SPI_CONFIG_OPTION_PHA_1** … clock phase
  - **SPI_CONFIG_OPTION_DATA_SIZE_8 or _16** … data size in bits
  - **SPI_CONFIG_OPTION_PRE_DELAY(x)** … 0-255 micro seconds of data delay after the SSEL assertion
  - **SPI_CONFIG_OPTION_POST_DELAY(x)** … 0-255 micro seconds of SSEL de-assertion after the end of data transfer

Return

SPI port handle when successful. NULL in case of an error.

Remarks

The handle returned by this function is used in all subsequent calls to any SPI functions when accessing this SPI port.

3.3.2 SPI_Close

Prototype

```
int32_t SPI_Close (LPC_HANDLE hSPI)
```

Parameters

- **hSPI** … SPI port handle returned by SPI_Open

Return

Zero if successful. The error codes are defined in the LPCUSBSIO_ERR_T enumeration.

Remarks

Always close an SPI port handle when it is no longer needed.

3.3.3 SPI_Reset

Prototype

```
int32_t SPI_Reset (LPC_HANDLE hSPI)
```

Parameters

- **hSPI** … SPI port handle returned by SPI_Open

Return

Zero if successful. The error codes are defined in the LPCUSBSIO_ERR_T enumeration.

Remarks

Reset the SPI controller.

3.3.4 SPI_Transfer

Prototype

```
int32_t SPI_Transfer (LPC_HANDLE hSPI, SPI_XFER_T *xfer)
```

Parameters

- **hSPI** … SPI port handle returned by SPI_Open
- **xfer** … transfer structure containing the target address memory buffer pointers and sizes

SPI_XFER_T structure

- **device** … index SSEL signal activated during the transmission
- **length** … number of bytes to transmit and receive
- **txBuff** … bytes to be transmitted
- **rxBuff** … receive data buffer
- **options** … additional transfer option flags, currently unused

Return

The number of bytes physically transferred. Negative status code in case of a failure.

Remarks

Perform a bidirectional SPI transaction. For more information about SPI data transfers, see Data transfer with a SPI target device. The last version of the USBSIO library does not support the communication options.

3.4 GPIO configuration and control functions

This section describes the functions used for the GPIO port and pin control.

- **int32_t GPIO_ReadPort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t *status)**

  reads a GPIO port.
- **int32_t GPIO_WritePort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t *status)**

  writes to a GPIO port.
- **int32_t GPIO_SetPort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)**

  sets the GPIO port bits.
- **int32_t GPIO_ClearPort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)**

  clears the GPIO port bits.
- **int32_t GPIO_GetPortDir (LPC_HANDLE hUsbSio, uint8_t port, uint32_t *pins)**

  reads the GPIO port direction bits.
- **int32_t GPIO_SetPortOutDir (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)**

  sets the GPIO port pins direction to output.
- **int32_t GPIO_SetPortInDir (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)**

  sets the GPIO port pins direction to input.
- **int32_t GPIO_SetPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)**

  sets a specific GPIO port pin value to high.
- **int32_t GPIO_GetPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)**

  reads the state of a specific GPIO port pin.
- **int32_t GPIO_ClearPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)**

  clears a specific GPIO port pin.
- **int32_t GPIO_TogglePin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)**

  toggles the state of a specific GPIO port pin.
- **int32_t GPIO_ConfigIOPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin, uint32_t mode)**

  configures the IO mode for a specific GPIO port pin.

3.4.1 GPIO_ReadPort

Prototype

```
int32_t GPIO_ReadPort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t *status)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **status** … pointer to a variable that receives the GPIO port value

Return

Number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Read the GPIO port. Use the status variable to return the state of all port pins as a single 32-bit value.

3.4.2 GPIO_WritePort

Prototype

```
int32_t GPIO_WritePort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t *status)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **status** … pointer to a variable which contains the GPIO port value to be written. This variable is updated with the new port value after it is updated.

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Write the GPIO port and update the state of all port pins configured as output. Use the status variable to return the new state of all port pins as a single 32-bit value.

3.4.3 GPIO_SetPort

Prototype

```
int32_t GPIO_SetPort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pins** … mask of pins to set to a logical high state

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Set all pins that match the bits set in the mask value to a high logical state. Other pins are not modified.

3.4.4 GPIO_ClearPort

Prototype

```
int32_t GPIO_ClearPort (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pins** … mask of pins to clear to a logical low state

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Clear all pins that match the bits set in the mask value to a low logical state. The other pins are not modified.

3.4.5 GPIO_GetPortDir

Prototype

```
int32_t GPIO_GetPortDir (LPC_HANDLE hUsbSio, uint8_t port, uint32_t *pins)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pins** … pointer to a variable which receives the pin direction value. The bit set to 0 refers to the input, the bit set to 1 refers to the output direction.

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Use this function to determine the current port input or output direction of all pins.

3.4.6 GPIO_SetPortOutDir

Prototype

int32_t GPIO_SetPortOutDir (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pins** … mask of pins to set as output direction

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Use this function to set the selected pins as the GPIO output. The other pins are not modified.

3.4.7 GPIO_SetPortInDir

Prototype

```
int32_t GPIO_SetPortInDir (LPC_HANDLE hUsbSio, uint8_t port, uint32_t pins)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pins** … mask of pins to set as input direction

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Use this function to set the selected pins as the GPIO input. The other pins are not modified.

3.4.8 GPIO_SetPin

Prototype

```
int32_t GPIO_SetPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pin** … index of a pin (0..31) to set

Return

The number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Use this function to set a single selected pin to a logical high state.

3.4.9 GPIO_ClearPin

Prototype

```
int32_t GPIO_ClearPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pin** … index of a pin (0..31) to clear

Return

Number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Use this function to clear a single selected pin to a logical low state.

3.4.10 GPIO_TogglePin

Prototype

```
int32_t GPIO_TogglePin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pin** … index of a pin (0..31) to toggle

Return

Number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

Use this function to toggle a logical value of a single selected pin.

3.4.11 GPIO_GetPin

Prototype

```
int32_t GPIO_GetPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pin** … index of a pin (0..31) to get

Return

Pin value of 0 or 1. This function does not return an operation error code.

Remarks

Use this function to read a value of a single selected pin.

3.4.12 GPIO_ConfigIOPin

Prototype

```
int32_t GPIO_ConfigIOPin (LPC_HANDLE hUsbSio, uint8_t port, uint8_t pin, uint32_t mode)
```

Parameters

- **hUsbSio** … SIO port handle returned by LPCUSBSIO_Open
- **port** … GPIO port number
- **pin** … index of a pin (0..31) to configure
- **mode** … pin mode as a PIO register value

Return

Number of bytes physically transferred. In this case, it is equal to 4 when successful. Negative status code in case of a failure.

Remarks

This function sets the pin IOCON configuration register (PIO register) to a specified value. This operation enables you to reconfigure the pin function, set the open-drain mode, and enable internal pull-up or pull-down resistors connected to the pin.This is an advanced function and it is not used in typical scenarios. Use it to reconfigure a pin normally used as SPI or I2C to a GPIO function. A value of 0x100 sets the pin to the Digital GPIO mode. A value of 0x300 also enables the open-drain mode, which is needed to work correctly with the I2C bidirectional level shifters.Using a library function that causes an SPI or I2C operation reconfigures all related pins back to the communication mode (MCU Link bridge only).See Accessing GPIO Ports and Pins for more information.

IMPORTANT: Use this function with caution. A wrong mode value may set the pin to an invalid state and lead to physical hardware damage in extreme cases. Always consult the mode change with the board schematic and check for potential hardware conflicts.

Chapter 4
Revision history

Table 1. Revision history

Revision number	Date	Substantive changes
0	27 July 2021	Initial release

Date of release: 27 July 2021

Document identifier: LIBUSBSIOUG